1997

Teamwork in genetic programming

Michael LaLena

Michael LaLena

Masters Thesis

Teamwork in Genetic Programming

Rochester Institute of Technology

School of Computer Science and Technology

Teamwork in Genetic Programming

by

Michael LaLena

A thesis, submitted to

The Faculty of the School of Computer Science and Technology

in partial fulfillment of the requirement for the degree of

Master of Science in Computer Science

Approved by: _____

Prof. Anderson

_____

Prof. Biles

_____

Prof. Wolf

April 28, 1997

Thesis Reproduction

I, Michael LaLena, hereby **grant permission** to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part.
Any reproduction will not be for commercial use or profit.

Date: May 8, 1997 Signature: _____

Abstract

This thesis attempts to solve food collection problems using genetic programming. The genetic program will evolve programs that mimic the way ants can collect food and bring it back to a nest. There are two special factors in this genetic program that make the ants work together in order to solve the problem, as opposed to each ant acting on its own.

First, there will be a stream in the environment that the ants must cross to get to the food. Although all ants have the same program, some must move into the water and die, building a bridge for the other ants to cross. The surviving ants must realize that a bridge has already been built that they can use, instead of killing themselves by building another bridge.

Second, the food will be too heavy for one ant to lift alone. The ants must find the food, and call to other ants for help. If all of the ants are at food waiting for help, some, but not all, of the ants must realize that they are in a deadlock situation, and leave their food to help other ants.

Both of these problems require the ants to use teamwork to solve the problem. The ants must realize what other ants are doing, without direct communication or a state machine within the ants.

Keywords

Genetic, Programming, Artificial, Life, Emergent, Behavior, Food, Collection, Ant

Computing Review Subject Codes

I.2.6 Artificial Life - Learning

I.2.2 Artificial Life - Automatic Programming

I.6.1 Simulation Modeling - Simulation Theory

I.6.3 Simulation Modeling - Applications

Table of Contents

1 Introduction

1.1 Overview

Genetic Programming is a means of evolving solutions to difficult problems, where the answer is not obvious. Genetic Programming allows problems to be solved without explicitly programming the solution.

This is done by way of a fitness function. The fitness function rates the performance of a possible solution. Good solutions are combined with other good solutions to hopefully create even better solutions. A genetic program starts with a set of functions, and continually combines the good functions, replacing the bad functions with the newly created ones. By this process, the genetic program can evolve a solution.

This thesis is about using genetic programming to create programs that mimic the food collection behavior of ants. The thesis will include a program to test how ants collect food. Many other people have written genetic programs to solve the food collection problem. The ant problem is almost a benchmark. My thesis will different from other genetic programs in one main way. I am concentrating on food collection problems where the ants must work together in order to succeed.

Most ant problems include many ants searching for food and bringing it back to a nest. The same solution that works for 20 ants will also work for one ant. 20 ants are used to solve the problem 20 times faster.

My program will be used to solve problems slightly different in that what 20 ants can do to solve a problem, 1, or even 19, will not be able to do to solve the same problem with the same solution. The ants will have to use teamwork in order to solve the problem.

I used two variations of the food collection problem. The first is adding water to the environment where the ants will collect food. By putting a lake of water between the nest and the food, the ants will have to figure out how to go around the water to move the food to the nest. This does make the problem more difficult, but it still does not require more than one ant. Placing a stream in the environment, that completely separates the ants from the water, will cause the need for more than one ant. As in nature, the ants will be forced to build an "ant bridge" to cross the water. Some of the ants will have to move into the stream and die to build a bridge for the other ants carrying food. The difficulties in this are having the ants with food find the bridge, and not having all the ants run into the stream and die.

The second variation is to have food that can only be lifted by more than one ant. Several ants will have to lift each piece of food and bring it back to the nest together. Not only do the ants have to work together, but must also find a way to communicate the need for help. There is also the problem of all of the ants waiting at different pieces of food for other ants to help lift. If there are three ants total, each individually trying to lift a different piece of food that requires two ants to lift, then nothing will get done. A way must be found for 1 or 2 (not all) ant(s) to abandon his food and help a different ant. This is extremely difficult since all the ants are running the same program.

1.2 Terminology

In this document, there will be many references to the 3 types of programs. To simplify understanding, these will be referred to as the genetic program, the ant programs, and the display program. The genetic program is the program which evolves the ant programs. The display program, or interface program is a separate executable that can visually execute one ant program.

All statements in the ant programs will be typed in all CAPS. Statements beginning with IF will be conditions. Tabs will be used to show the relationship between conditions and their clauses. The if and else clause statement blocks for the condition will be at least 1 tab out from the condition statement.

All parameters to the program will be in *ITALIC CAPS*. These parameters are all treated as integer defines for all of the source code. They determine how the genetic program will run, and are provided to the program in a params file.

An iteration is defined as one round of crossover, mutation, and fitness evaluation. Every iteration, 2 new ant programs are produced by the genetic program.

A turn is defined as every ant executing the ant program once in a fitness evaluation. The fitness algorithm will run for hundreds of turns to evaluate the ant program.

Pheromones are a scent that ants can release to mark an area or a path. The same ant, or other ants can pick up the scent and act based on the direction of the scent.

2 Genetic Algorithms

2.1 Definition

Genetic Algorithms are a means of solving problems without explicitly stating the solution. This is done using the idea of Survival of the Fittest. Good solutions are combined with each other, to hopefully create a better solution. The Genetic Algorithm can be broken up into 5 different steps as shown in the following pseudo code:

```
Initial Population
Fitness Evaluation
WHILE (best solution not found)
{
        Tournament Selection
        Crossover
        Mutation
        Fitness Evaluation
}
```

When the Genetic Algorithm is first executed, an Initial Population of solutions is created. These are random solutions to the problem, and most of them will not be good. Some will even do the opposite of the desired effect.

Fitness Evaluation refers to testing the solution's ability to solve a problem. This is done so that decisions can be made on whether one solution is better than another. Solutions that can partially solve a problem are given a better fitness than those that don't.

In the Tournament Selection stage, a small number of solutions are picked from the population of solutions. Based on the fitness of these solutions the best two are chosen for Crossover. These best two solutions are the parents.

The Crossover operation takes the two parents from the tournament selection, and combines them to create two new solutions, the children. The goal is that the crossover will produce at least one child that is better than both of the parent solutions. The two new children will replace the two worst solutions from the Tournament Selection. This will keep the population the same size.

The Mutation operation makes a small change to one, both, or none of the children randomly. The change is done for two reasons. First, it might create a better solution that couldn't easily be found by performing a crossover operation. The other, and more important reason, is that it keeps a fresh supply of genetic material in the population. If the crossover operation is done to the population long enough, without mutation, then eventually every solution in the population will be identical.

A special type of Mutation is Greedy Mutation. In this case, if the mutated child is not a better solution than the unmutated child, then the mutation is undone. Because greedy mutation does not negatively affect the population, the rate of mutation can be increased when using Greedy Mutation.

In the pseudo code on the previous page, the Genetic Algorithm will run until the best solution is found. This is not always the case. Many times the Genetic Algorithm will find a very good solution, but not the best one. Also, if the Genetic Algorithm is run for a long enough period of time, all of the strings in the population will look the same. To some degree this effect can be countered with Mutation or a larger Population, but not completely.

The number of strings in the population will have an effect on how fast the Genetic Algorithm can find a solution. With a small population, a good solution will be found quickly. With a large population, it will take longer to find a good solution, but it will have a better chance of finding a great, or the best, solution.

The reason for this is simple. To quickly find a solution, the genetic program needs to use the best strings in the population to create a better solution. The best strings are probably the children of a previous crossover. If the population is smaller, then these children will be reused to make more strings more quickly. The problem is that with a small population the Genetic Algorithm takes a narrow minded approach to solving the problem. Instead of having many dissimilar solutions that have a good fitness, there are only a couple of solutions with a good fitness, and they are very similar. The Genetic Algorithm will quickly find the best solution it can get with what it has to work with. Unfortunately, there is not much to work with, and the solution will not be great.

2.2 Example

A simple example of a Genetic Algorithm is to create a string of all 1's. The initial population will be six random solutions (strings) of length 5, where each element of the string is a 1 or 0. The fitness will be the number of 1's in the string. The worst fitness is 0, and the best fitness is 5.

String	Fitness
1 1 0 1 0	3
0 1 0 1 0	2
1 1 0 0 0	2
0 1 0 1 1	3
1 0 1 0 1	3
0 1 0 0 0	1

After the fitness of the initial population is known, a small group of strings are selected from the population, and the two with the best fitnesses are chosen. Say that solution 1, 2, 5, and 6 are chosen. The best two are 1 and 5, both with a fitness of 3.

In Genetic Algorithms, the most common form of crossover is to pick a crossover point in the parent strings, and split the parent string into two parts. Say for this example, the crossover point is after the second element. Then, the first child is the first half of parent 1 and the second half of parent 2, and the second child is the first half of parent 2 and the second half of parent 1.

		Fitness
PARENT 1	1 1 0 1 0	3
PARENT 2	1 0 1 0 1	3
CHILD 1	1 1 1 0 1	4
CHILD 2	1 0 0 1 0	2

One of the children has a better fitness than both parents, and one is worse. If mutation was being performed, then the next step would be to possibly toggle one of the numbers in each child. Since this is a simple problem to solve, assume that neither child is mutated.

Now the two children replace the two worst solutions from the Tournament Selection. Child 1 replaces the solution in position 2, and child 2 replaces position 6. Now, the average fitness of all solutions has risen, and the value of the best solution in the population has also risen. The perfect solution of 5 1's can be found with just one more Tournament Selection.

2.3 Encoding

The method by which the solutions are encoded into strings is very important. In the above example, the simple direct encoding of the strings will work. In other examples, it will not.

Consider the Travelling Salesman Problem. In this problem, the goal is to find the shortest distance between N cities, and back home. The fitness would be the distance of the path, with the shortest path being best. This cannot be simply encoded as the order in which the cities are travelled. Example:

PARENT1	1 2 3 4 5
PARENT 2	5 4 3 2 1
CHILD 1	1 2 3 2 1
CHILD 2	5 4 3 4 5

With straight order encoding, normal crossover cannot be performed. The crossover will have to have built in checks to prevent cities from being repeated in the list. If this is done, then the simplicity of the Genetic Algorithm is lost.

Another method is to use the city numbers as slots as below.

The city tour:	1 4 2 6 5 3
is stored as:	1 3 1 3 2 1

In the encoding, each number represents the city number of the city minus the number of cities before that number that have already been visited.

X represents a visited city. O represents an unvisited city.

```
O O O O O O
X O O O O O       City 1 selected. 1st free slot.
X O O X O O       City 4 selected. 3rd free slot.
X X O X O O       City 2 selected. 1st free slot.
X X O X O X       City 6 selected. 3rd free slot.
X X O X X X       City 5 selected. 2nd free slot.
X X X X X X       City 3 selected. 1st free slot.
```

This encoding has the property that element X in a string of length N will be less than or equal to N-X+1. If two parents strings have this property, then if the standard crossover is used, both children strings will also have this property. This means that the children strings are valid solutions. Also, the last position in the string is always 1. This means that the string is 1 number smaller than the number of cities. This decreases the search space for the Genetic Algorithm.

This encoding method also makes mutation easier. Only one position needs to be changed randomly to mutate the path. In the previous example, two cities would have to have their position switched.

This encoding method has one major flaw. The most important attribute of a possible solution is the order in which the cities are travelled. A good solution may only have 2 out of 10 cities in the wrong order. With the slot encoding, the entire city order can be lost during a crossover.

The encoding: 1 3 1 3 2 1

is the city tour: 1 4 2 6 5 3

while encoding: 5 3 1 3 2 1

is the city tour: 5 3 1 6 4 2

There are 6 city adjacencies in a 6 city tour. Although only one number changed in the tour encoding, 4 of the 6 city adjacencies where changed: 5-3 and 4-2 stayed the same. Thus, this is also a bad method.

A better, but more complicated way to store the tour is to directly store the city adjacency list. The number at position X in the string represents the city that X connects to.

Thus, the tour: 1 4 2 6 5 3

is stored as: 4 6 1 2 3 5

and the tour: 1 4 6 2 5 3

is stored as: 4 5 1 6 3 2

In these two tours, only the 6 and 2 are reversed, and only 3 out of the 6 adjacencies are different. The standard crossover operation will not work with this example, for the same reason it doesn't work with the straight city list.

A Greedy Crossover Operation will work here. All elements for both encodings that are the same are moved into both children from the parents. Thus, both children will start as: 4 X 1 X 3 X. The first X in slot 2 is filled in for Child 1 by taking slot 2 from Parent 1. The first X in slot 2 is filled in for Child 2 by taking slot 1 from Parent. The second X slot is filled in for each child by the opposite parent. This is difficult, because the crossover operation must make sure no numbers are repeated in each child, while also trying to keep each child different.

PARENT 1: 4 6 1 2 3 5

PARENT 2: 4 5 1 6 3 2

CHILD 1: 4 5 1 2 3 6

CHILD 2: 4 6 1 2 3 5

When a conflict arises, it is best to take a number from one parent, before picking a random city number that has not been used. Mutation for this method requires two numbers to be switched. The Mutation can be improved by allowing only positions that the parents didn't agree on to be mutated.

This is a good encoding method, but still has one minor flaw.

The city tour: 1 2 3 4 5 6

is encoded as: 1 2 3 4 5 6

while the tour: 6 5 4 3 2 1

is encoded as: 6 5 4 3 2 1

Both of these tours are identical, but one is in the opposite order of the other. Thus, the Greedy Crossover will see no similarities in these identical solutions. A better method of encoding would store city adjacencies in both directions, either as two city lists, or a 2 dimensional adjacency bit map. Either method would make crossover extremely difficult.

The Travelling Salesman Problem shows that selection of an encoding method, and crossover method can be critical to solving the problem.

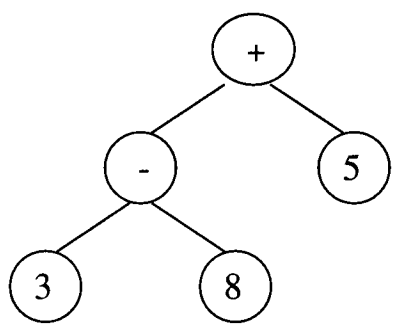
3 Genetic Programming

3.1 Definition

Genetic Programming is simply Genetic Algorithms where the solution strings are replaced with variable length programs. The goal is to find a program that can solve a desired problem. The program is evaluated for fitness by executing the program.

Because the population in a Genetic Program is variable length solutions, it is best to store the solutions in a tree format. A tree structure also works well with storing and executing programs, especially those with conditions. For this reason, LISP is a good language for Genetic Programming. LISP stores all data in tree structures, and can execute the tree data structure if it contains program instructions.

The Language that the solutions use can be a variety of types. The simplest language is mathematics. In this situation, the leaf nodes of the tree are numbers or variables, and the non-leaf nodes are operators: +, -, *, /, %... An example mathematical tree would be:



The next level of Genetic Programming Languages are those languages that have the same statements as the language being programmed in. This would include assignments, conditionals, and loop constructs. This might also include calls to special prewritten functions. This type of Genetic Program is the most suitable to LISP.

In the final language type, no similarities exist between conventional programming language and the Genetic Programming Language. This could be implemented in LISP as all function calls. It could also be implemented in C, or any other conventional language, by writing an intepreter. The intepreter would take the program tree and traverse the tree, acting on each statement as it reaches it.

Genetic Programming follows the same steps as Genetic Algorithms, but the steps act a little differently.

The solutions in the Initial Population of the Genetic Program will vary in size. It is desirable to have a range of program sizes, while making sure that the computer does not run out of memory. Assuming that there is some sort of condition statement in the language,

or an operator like + that requires parameters, then there will be a difference between leaf nodes and non-leaf nodes. When creating the initial population, care must be taken to create a legal program.

In the Fitness Evaluation stage, the program must be executed to determine how well it works. Depending on the application, the program may be executed more than once to determine fitness. Say that the Genetic Problem is being used to simulate the food collection process of ants. The goal is to have the ant collect all of the food within 100 turns. The program solution determines how the ant will act for each of the 100 turns. The program must be executed 100 times to determine fitness. If there are multiple ants in the problem, then the program must be executed 100 times for each ant. While determining the fitness, the Genetic Program must keep track of the state of each ant and changes to the pseudo environment that they are in.

Tournament Selection works exactly the same for a Genetic Program as a Genetic Algorithm.The Tournament Selection operation may look at the same program many times before it is replaced. In an Genetic Program, most of the CPU time is spent testing the fitness of the programs. That is why the fitness is calculated after each program is made, not when the fitness is needed in Tournament Selection.

For Crossover, a crossover node in each parent program tree must be selected. Unlike Genetic Algorithms, the crossover points in both parents do not have to be the same, and probably cannot be. It is likely that the parents will have a different number of nodes, and have a different configuration.

To be truly random, the crossover node must have an equal chance of being any node in the parent tree. This would be done by finding the number of nodes in the tree N, picking a random number R from 1 to N, and then doing an preorder traversal of the tree through R-1 nodes.

A truly random crossover node selection may not be desired. Maybe, towards the end of the Genetic Program, only minor changes to the parent programs are desired. This can be done by making it more likely that a leaf node of the parent tree will be selected as the crossover node. This way, only one node changes from parent to child. On the other hand, to make large changes to the program, make it more likely a non-leaf node will be selected as the crossover node.

Usually, in a Genetic Program, the leaf nodes and non-leaf nodes are not interchangeable. As in the mathematical example, the leaf nodes were operands and the non-leaf nodes were operators. This property must be preserved during Mutation. A leaf node can only be mutated into another legal leaf node, and a non-leaf node can only be mutated into a legal non-leaf node.

3.2 Example

One of the most basic Genetic Programming examples is the food collection problem. In this problem, the program must guide an ant around an environment to collect food. The solution will be composed on statements that will move the ant around and search for food. The ant will continually execute the program each turn until all of the food is found.

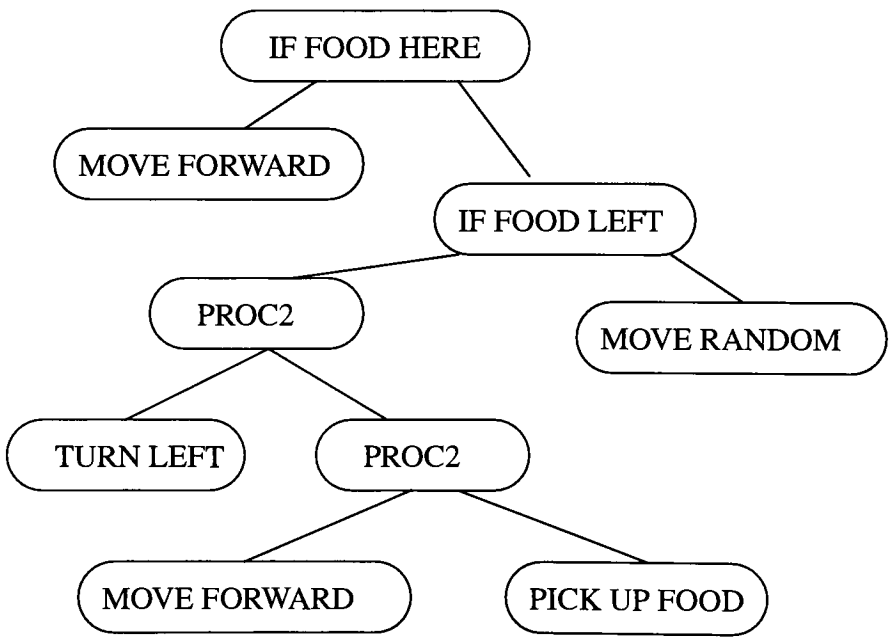
Assume that the programs are stored in a binary tree format. No node can have 1 child, just 0 or 2. In this problem, the leaf and non-leaf statements will be different. The leaf statements will be operations:

TURN LEFT
TURN RIGHT
MOVE FORWARD
MOVE RANDOM DIRECTION
PICK UP FOOD
DROP FOOD
NO OPERATION

These operations will allow the ant to move around and pick up and drop food. The non-leaf statements will be conditionals. In a condition, the left statement is executed if the condition is true, and the right statement is executed if the condition is false. Possible non-leaf statements are:

IF FOOD AHEAD THEN
IF FOOD LEFT THEN
IF FOOD RIGHT THEN
IF FOOD HERE THEN
PROC2

The PROC2 statement is not a condition. Both children statements of a PROC2 node are executed.



This example is a program tree. It shows how PROC2 statements can be useful. If there is food to the left, then three statements need to be executed:

TURN LEFT
MOVE FORWARD
PICK UP FOOD

Two PROC2 statements allow the single left statement of the IF FOOD LEFT condition to be turned into 3 statements.

The MOVE RANDOM statement simply moves the ant in a random direction to simulate a random search for food.

Usually, not all of these statements would be used in the same Genetic Program. A perfect solution could probably be found just using half of the statements listed. Although the solution would be slightly more complicated, because of the decrease in the number of possible statements, the search space for the perfect program will be a lot smaller. Therefore, the Genetic Program should find a solution faster.

Obviously, when creating the initial population, statements like IF FOOD HERE cannot be leaf nodes of the tree. The Genetic Program must make sure that all of the programs in the initial population are legal programs.

The programs in the initial population will also vary in size. One method of creating the initial population is to recursively pick a statement from the list of possible statements, and make that the next statement. If the statement is a non-leaf statement, then recursively create children nodes for that statement. If there are more than twice as many leaf statements as non leaf statements, then this method will work. Otherwise, the programs could start becoming infinitely big.

If this is the case, then the statement selection needs to be weighted in favor of leaf statements. How much the selection process favors leaf nodes will affect the average size of the program trees. If possible, the selection process could be weighted in the other direction to create bigger trees.

The program trees must be executed to evaluate them for fitness. If the Genetic Program is not written in LISP, then the Genetic Program must have an intepreter to execute the program. The steps to evaluate the program are:

Set Up the Environment to run the simulation in.
***WHILE* (Goal Not Achieved) *AND* (Not Max Turns)**
{
    Interpret program once.
}

In this example, to set up the environment, the pieces of food are placed at their start positions, and the ant is placed at its start position. If the environment is stored as a 2-dimensional array, then this step is done by simply copying the data from a master array into the temporary one.

The food and ants cannot be placed randomly on the map. Doing this could cause the fitness of a program to change each time it is executed. A more stable environment is needed to find the best solution. Depending on the application, it may also be desirable to start with the same random number generator seed every time a program is executed.

This intepreter will run until the ant has found all of the food. This is the goal. The intepreter must also make the assumption that the program might not get the ant to find all of the food, no matter how much time the ant is given. If the program does not contain the MOVE FORWARD statement, or the PICK UP FOOD statement, then the ant cannot get any food. For this reason, the program is usually only executed a set number of times, or turns.

The intepreter will recursively traverse the program tree to execute it. Basically, there are two main actions for the intepreter:

```
void statement(node)
{
        if (leafNode(node))
        {
                interp(node)
        }
        else
        {
                if (node.type == PROC2)
                {
                        statement(node.left)
                        statement(node.right)
                }
                else
                {
                        if (eval(node))
                        {
                                statement(node.left)
                        }
                        else
                        {
                                statement(node.right)
                        }
                }
        }
}
```

The code for the intepreter is very simple. Two main functions need to be written for interpreting each statement. **interp()** will execute one leaf statement, and **eval()** will evaluate one non-leaf statement. Basically, both of these function will be one large case statement for each node type.

The example program tree from above can be written as:

```
                IF FOOD HERE
                        MOVE FORWARD
                ELSE
*                       IF FOOD LEFT
-                               PROC2
                                        TURN LEFT
                                        PROC2
                                                MOVE FORWARD
                                                PICK UP FOOD
                        ELSE
                                MOVE RANDOM
```

Another sample program could look like:

```
                IF FOOD RIGHT
                        PROC2
                                PROC2
                                        TURN RIGHT
                                        MOVE FORWARD
                                PICK UP FOOD
                ELSE
                        IF FOOD LEFT
*                               PROC2
                                        TURN RIGHT
                                        PROC2
                                                MOVE RANDOM
                                                PICK UP FOOD
                        ELSE
                                TURN LEFT
```

Say that these two programs are the parents of a crossover operation. The * marks the crossover node in each program, and the -'s are the nodes that are children of the crossover node. The crossover operation would product the following children programs:

```
                IF FOOD HERE
                        MOVE FORWARD
                ELSE
*                       PROC2
                                TURN RIGHT
                                PROC2
                                        MOVE RANDOM
                                        PICK UP FOOD
```

and

```
IF FOOD RIGHT
        PROC2
                PROC2
                        TURN RIGHT
                        MOVE FORWARD
                PICK UP FOOD
ELSE
        IF FOOD LEFT
*               IF FOOD LEFT
                        PROC2
                                TURN LEFT
                                PROC2
                                        MOVE FORWARD
                                        PICK UP FOOD
                ELSE
                        MOVE RANDOM
        ELSE
                TURN LEFT
```

The first child probably has a bad fitness. This can be found out by simply looking at the program. There is no reasoning in the ordering of the statements. In some instances, the ant would run away from food.

The second child program seems to be much better. It gained the ability to find food to the left from the first parent program, and the ability to find food to the right from the second parent. Unfortunately, the ant cannot move if there is not food next to it. It does not execute a MOVE FORWARD, unless it is moving to food it knows about. It does have a MOVE RANDOM statement, but this statement can never be executed. The first IF TURN LEFT would have to be true, while the second IF TURN LEFT condition would have to be false. If the last statement was some kind of MOVE statement, then this would be a good solution. Changes to the program from mutation might be able to make this fix.

A very good solution would look like the program below. PROC2 statements have been removed for readability. It can be assumed that if two consecutive statements are on the same tab level, they have a hidden PROC2 statement as their parent.

```
IF FOOD LEFT
        TURN LEFT
ELSE
        IF FOOD RIGHT
                TURN RIGHT
        ELSE
                PICK UP FOOD
IF FOOD AHEAD
        MOVE FORWARD
        PICK UP FOOD
ELSE
        MOVE RANDOM
```

This program is a more concise solution than the children of the crossover operation. There are two PROC2 statements in this program. One connecting the IF FOOD LEFT and IF FOOD AHEAD statements on the first level, and one connecting the two statements in the true clause of IF FOOD AHEAD.

Note that the IF FOOD HERE statement was not used. If there is food at the current location, and there is no food to the left or right, then the food is picked up by the first PICK UP FOOD statement.

This problem can be improved by using multiple ants. For this, each ant will execute the program once per turn, one ant after another. Each ant would be changing the environment before the next ant ran the program. Using this food collection example, adding more ants would allow all of the food to be found faster.

3.3 Encoding

All of the Genetic Programs discussed have stored solutions in binary trees. This is done to simplify the writing of the Genetic Program. An encoding method that allowed a variable number of children would remove the need for nested PROC2 statements. One PROC3 or PROC4 statement could be used instead to connect 3 or 4 children respectively. Non-binary trees would also allow for conditions that had more that two results, and also case statements. While these are useful constructs in a real programming language, they can be simulated with binary tree programs to achieve the same effect. Under the belief that simpler is better, this encoding discussion will be limited to binary trees.

Since the encoding method is limited to binary trees, all of the encoding methods will be limited to statement or instruction set selection. When selecting the instruction set, the less number of different statements used the better, as long as enough different statements are available to solve the problem.

The simplest example of a Genetic Program is the mathematical program. Here, there are only 5 main operators: +, , /, *, %. Other more complex operators could be added, but they probably wouldn't be necessary. Assuming that negative number operands are allowed, the - operator is not necessary to solve the problem. Also, assuming real numbers are used, / is not needed.

Now, the operators are limited to +, *, and %, and the operands are signed real numbers. This cuts the number of operators by 40%, which will have a similar affect on the size of the search space.

Division and Mod are both protected operations. This means that if the second operand is 0, the Genetic Program will not allow a mod or divide by zero. The result of a mod or div by zero will be 0. Since one of these protected operations, the divide, is one of the unnecessary statements, then this also simplifies the Genetic Program.

In the ant program example above, a NO OPERATION statement was listed. A no-op statement might be desired in some cases. If the ant only wants to act on the if clause of a statement, then the no-op could be the else clause. But, a special NO OPERATION statement is not needed. In this example, PICK UP FOOD is also a no-op statement. When a no-

op statement is needed in an if or else clause, PICK UP FOOD can be used. It will not turn or move the ant. Even if it is already known that there is no food to pick up, the PICK UP FOOD statement can be executed. It will just have no effect.

The IF FOOD HERE statement is also not necessary. The ant could execute a PICK UP FOOD statement every time it moved, without looking to see if food was there.

The IF FOOD LEFT and IF FOOD RIGHT statements could be removed. The ant could turn in that direction, and then execute a IF FOOD AHEAD instead.

Technically, TURN RIGHT could be replaced with three consecutive TURN LEFT's, but this would not be a solution that was found randomly.

Thus, 6 statements take the place of 12, cutting down the complexity of the search space.

More important than selecting an instruction set is deciding how condition statements will work. Will the condition statements have an else clause? Will the inverse of condition statements be allowed? Will the conditions have side effects?

Conditions do not need to have an else clause. To keep the program in a binary tree, both children of the condition nodes would be executed if the condition was true. In very simple problems, this might be desired.

Condition statements could also have their inverses. If the condition IF FOOD AHEAD is in the instruction set, then IF NOT FOOD AHEAD could also be added. If the conditions have else clauses, then this is not necessary. The inverse of the condition exists already by switching the if and else clauses of the condition.

Adding side effects to a condition will greatly improve the ability of the Genetic Program to find a solution to the problem. A side effect is an action that is done before the if or else clause of the condition. The action can be done before or after the condition is evaluated, or the condition can be the whether the action was successful.

Example side effects are:

Without side effects	With side effects
IF FOOD HERE PICK UP FOOD MOVE TO NEST	IF PICK UP FOOD MOVE TO NEST
IF NEXT TO FOOD MOVE TO FOOD PICK UP FOOD	IF MOVE TO ADJACENT FOOD PICK UP FOOD or IF PICK UP ADJACENT FOOD MOVE TO NEST
IF AT NEST DROP FOOD MOVE RANDOM	IF DROP FOOD AT NEST MOVE RANDOM

These side effects will simplify the instruction set, and the final solution. Food will be picked up and dropped automatically, so these statements can be removed from the instruction set. For the PICK UP ADJACENT FOOD example, the ant checks to see if it is next to food, turns toward the food, moves to the food, and picks it up, all as a side effect. The if and else clauses can act on whether the ant did pick up the food or not.

These side effects make certain assumptions. In these cases, they assume that if the ant finds food, it wants to pick it up. If there is a better solution where the ant does not pick up the food when it finds it, a Genetic Program using these side effects would not be able to find it.

This can be shown in the following multiple ant food collection problem, where the food is set up in piles, ant the ants are carrying it back to the nest. Each ant can carry only one piece of food.

```
IF CARRYING FOOD THEN
        MOVE TO NEST
ELSE
        IF NEXT TO FOOD OR FOOD HERE THEN
                IF ANOTHER ANT HERE THEN
                        PICK UP FOOD
                        MOVE TO NEST
                ELSE
                        CALL FOR ANOTHER ANT
        ELSE
                IF I HEAR ANOTHER ANTS CALL THEN
                        MOVE TOWARD CALL
                ELSE
                        MOVE RANDOM
```

There are several steps to this problem.

1) Move around until one ant finds a piece of food.

2) After the first ant finds food, he calls other ants, and all ants move toward that ant.

3) If other ants also find food on the way to the first ant, they also call ants. Ants are attracted to the closest call.

3) When an ant meets a calling ant, the calling ant picks up food and brings it to the nest

4) If the ant that met the calling ant is next to food (likely since the food is in piles), then that ant starts calling other ants.

In this example, ants do not pick up food immediately. The ants do not want to forget where the pile of food is, so they wait for another ant to mark the spot. Then, they take a piece of food to the nest. If the ant automatically picked up food and started moving to the nest with it, this solution would not exist. Perhaps, no solution as good as this one could be found.

Another important part of this program is the second MOVE TO NEST, after the PICK UP FOOD. If this statement were not there, then both ants would leave with food. First, an ant would answer the call. When that ant arrived, the ants would see each other, and both would pick up food. On the next turn, both would see that they have food, and bring it to the nest.

4 Schemata

4.1 Partial Solutions

Simply put, a schema (plural: schemata) is a partial solution to a genetic problem. The solution is encoded in a string, and the parts of the string that contribute toward solving the problem are schemata. In order to solve the problem, these schema must be preserved during the crossover operation.

In the previous Travelling Salesman Problem, the schemata would be a section of the shortest path. In the encoding discussion of the Travelling Salesman Problem, it was shown that certain encoding methods preserved parts of the parent strings better than others. The reason this was important is that it also allowed the schemata to be preserved from parent to child.

For example, say the shortest tour is:

1 2 3 4 5 6 7 8 9

and the parent strings are:

2 3 6 8 9 1 4 5 7

and 6 7 4 3 1 8 2 9 5

In the first parent, the schemata are 2-3, 8-9, and 4-5. In the second parent, there is one three-city schema: 5-6-7, which wraps around the end of the string. Also, 4-3 can be considered a good schema, if the encoding method can recognize a backwards link. The encoding and crossover methods should allow as many of these schemata to be passed to the children as possible.

Four encoding methods were discussed for the Travelling Salesman Problem. The first was a direct encoding of the city numbers. After performing a crossover, only one city adjacency will be separated in each parent, so at most one schemata will be disturbed. However, the child will most likely not be a legal solution. Some city numbers will be repeated and others will be missing. Correcting this will disrupt many more schemata.

The second encoding method was the free slot method. Instead of putting the city number in the string, the number of unused cities before the next city is put in the string. This method removed the special case checking for the crossover operation, but it was shown that a single crossover could destroy all schemata in the string.

The other two encoding methods stored city adjacencies (single and bi-directional), and used a greedy crossover. The bi-directional adjacency list recognizes twice as many schemata as the single adjacency method. Both methods for both children will preserve all schemata that exist in both parents. Most of the other schemata will be preserved in at least one of the children.

4.2 Schemata in Genetic Programming

In Genetic Programming, the schemata are portions of the program, instead of a substring of the solution. This partial solution is a series of statements that solve part of the problem. For the ant problem, some examples would be:

```
IF FOOD HERE
        PICK UP FOOD

IF FOOD LEFT
        TURN LEFT
        MOVE FORWARD
        PICK UP FOOD
```

The schemata can even be less complicated than this. In the second example, the and does not need to pick up the food for it to be a schema. Seeing the food to the left and then turning left is also a schema. The PICK UP FOOD statement alone is a schema. The inclusion of this statement in the program, whether it ever successfully picks up food or not, makes the program better than other programs without this statement. If the parents do not have the PICK UP FOOD statement, then without mutation, the children will never be able to solve the problem.

Preserving schemata in Genetic Programming is more difficult that Genetic Algorithms. There is no greedy form of crossover that is also simple. The most that can be done is to determine how high or low in the tree the crossover point will be. A crossover point high in the tree will be a include many nodes, while one low or at the bottom of the tree will include few or one node.

5 Search Space

5.1 Searching

Genetic Algorithms & Programs search for a solution to the problem. The search space where they look for the solution is the set of all possible strings or programs. The searching analogy can best be envisioned by having every solution in the search space graphed three-dimensionally, where the Z axis was the fitness of the solution at point (X,Y). Assuming that good solutions were near great solutions, it would look like a topography with hills and valleys. The hills would represent the collections of good solutions, and the valleys would be bad solutions

5.2 Blind

Blind searching refers to searching for a solution without regard for the results of previous attempts at searching. The two most basic forms of blind searching are exhaustive and random. In an exhaustive search, every element of the set is tested to see if it is the best. In the Travelling Salesman Problem, for a 10 city tour, this set size would be 10! = 3628800. This is the amount of different possible tours. For each of these 3628800 possible solutions, the search program must add 10 numbers together. Adding 1 city to the tour, making it an 11 city tour, will increase the size of the set by 11 times. It can be easily seen that this problem can get quickly out of hand. A large tour could take a computer eons to solve.

Random searching is just as bad. Based on how good and bad solutions are grouped, a random search may find a good solution faster than an exhaustive search, but it will take just as long to find the best solution.

This is the reason that Genetic Algorithms are used. They try to move toward the best solution within the search space. This is done by building upon the schemata that the Genetic Algorithm has already found. If schemata are not preserved, then this method is reduced to a blind search.

5.3 Hill Climbing

Assuming that only one solution is the best, then one hill will have a higher peak than all other hills. When the initial population is created, it will be randomly scattered in the topography. After many crossover operations, the population will start to move up the different hills in the topography, as better solutions are found. The goal is that some of the solutions will work their way up to the top of the highest hill.

When two solutions are on the same hill, then they are approaching the solution in the same way. That is, they will share schemata, because they are working toward the same good solution. When these two solutions were used in crossover, it may create a better solution in the same neighborhood. When two solutions are not on the same hill, the result of a crossover may be useless.

Take the example of the Travelling Salesman Problem. In this problem there can be two best solutions. The shortest path, and the shortest path in reverse. If one near perfect solution in one direction is combined with a near perfect solution in the other direction, most likely both children will have a very poor fitness. When two near perfect solutions on the same hill are combined, it could result in finding the best solution.

A small initial population will quickly converge to a solution, but will probably not find the best solution. The reason for this is hill climbing. If none of the initial solutions start at the base of the hill with the best solution, then the Genetic Algorithm will may find this hill at all. All of the effort of the Genetic Algorithm will be concentrated on a small percentage of the hills. This best solution of this group of hills will be found quickly, but not the best solution overall.

The number of perfect solutions can also effect the ability of the Genetic Algorithm to find a perfect solution. In some coding methods for the Travelling Salesman Problem, there may be 2N perfect solutions, where N is the number of cities. For both directions, there are N tours, depending on which of the different N cities is the first in the tour. In any coding method that preserves location in the list, instead of adjacencies, the schemata for each of the 2N perfect solutions will be completely different.

Hill Climbing in Genetic Programming is slightly different. Here, there can be many perfect solutions that are different, but they can all share the same schemata. One perfect solution might be the same as another perfect solution, but with an extra no-operation statement thrown in. Both programs will execute in the same way. In this case, there is no adverse effect in having many perfect solutions. It is actually beneficial.

6 Emergent Behavior

Emergent Behavior refers to the application of seemingly simple rules that lead to complex overall behavior. Many examples of Emergent Behavior can be found in nature, such as the actions of ants. If nature can evolve complex actions from a simple set of rules, then it makes sense that a Genetic Program can also evolve a solution to a complex task using a couple of simple computer instructions.

This Emergent Behavior may or may not consist of several individuals performing the same action. Some behaviors that are not possible with one individual may be possible with many individuals. Other behaviors are possible with one individual, but can be performed more efficiently with many.

This thesis will explore different food collecting behaviors in food collection problems. The basic problem will be a straight forward "bring the food back to the nest" problem. As more obstacles are added to the problem, hopefully the Genetic Program with the limited instruction set will be able to find an Emergent Behavior to solve the problem.

7 The Food Collection Problem

7.1 The Santa Fe Trail Problem

The most basic form of the food collection problem is the Santa Fe Trail Problem [Koza93]. This is a one ant problem. For this problem, the goal is to collect all of the food that is along a trail. There may be gaps in the trail, but if the ant loses the trail and continues forward, the trail will be picked up again.

This is a short form of the Santa Fe Trail (**S**=start, **E**=End, **x**=food, **-**=gap):

```
Sxxxxx-xx--xxx---x
                x       -xx--xxx---
                x       x           x
                -       x           x
                x-x--xx             E
```

Assuming food is picked up automatically, this problem can be solved with just three statements:

MOVE_FORWARD
TURN_LEFT
IF_NOT_FOOD_AHEAD (without an else clause).

The solution to the problem is:

```
IF_NOT_FOOD_AHEAD THEN
        TURN_LEFT
IF_NOT_FOOD_AHEAD THEN
        TURN_LEFT
IF_NOT_FOOD_AHEAD THEN
        TURN_LEFT
IF_NOT_FOOD_AHEAD THEN
        TURN_LEFT
MOVE_FORWARD
```

The solution is to simply spin around until food is directly in front of the ant, or the ant is facing the original direction, and then move ahead one space.

The best way to train ants to find this solution is to have the trail become increasingly more difficult as more food is found. Have the fitness be the amount of food found. The ants with the better fitness are those that can overcome more obstacles. The obstacles in this problem are: gap of 1 space, gap of 2 spaces, gap of 3 spaces, turn, food missing at turn, and food missing before and at turn.

The selection of an instruction set is also important to finding a solution. Most would assume that if a TURN_LEFT statement were included in the instruction set, then there would also be a TURN_RIGHT statement, but it is unnecessary. It would not simplify the solution. The only change is that the 2nd and 3rd TURN_LEFT statements would

become TURN_RIGHT statements. Adding a TURN_RIGHT statement would only increase the size of the set of possible ant programs that the genetic program must search to find a solution.

For the same reason, a IF_NOT_FOOD_AHEAD statement is desired, instead of a IF_FOOD_AHEAD statement, since a special action is only required if food is not in front of the ant.

7.2 The Single Ant Problem

The Single Ant Food Collection Problem deals with an ant collecting food that is set up in piles instead of being laid out along a trail. Normally, the Single Ant Problem is solved with several ants, but it is not required.

In this problem, the ants must find the food, and bring it back to one location. This location is usually the nest where the ants start. It is assumed that the ants know how to find the nest, but do not know where the food is, even after they have seen a pile of food.

Since the search for food is random, the food must be placed in piles, so that the entire problem is not random. Once the food is found, the ants can use pheromones to mark the pile of food, and the trail back to the nest. The ants can then follow the trail from the nest to the food. Other ants can stumble across the trail and follow it to the food.

If there are multiple piles of food, then it is necessary that the pheromone trail dissipate, so that once one pile of food is exhausted, no more ants will go in that direction. If the trail does not dissipate, then all of the ants will stay in the pheromone trail, and none will be able to escape to find another pile of food.

In this scenario, it is necessary for a IF_MOVE_TO_AWAY_PHEROMONE operation. Since the pheromones dissipate, the strongest pheromone scent will be directly behind the ant that is travelling to the nest with food. Ants cannot move to the strongest pheromone scent that is adjacent to them, because that will be in the direction of the nest, not the food. In the IF_MOVE_TO_AWAY_PHEROMONE statement, the ant moves to the strongest pheromone that is in one of the 2 directions away from the nest.

Again, selecting the instruction set is very important to solving the problem. Assuming food is automatically dropped when the ant is over the nest, the instruction set for this problem would be:

MOVE_TO_NEST
PICK_UP_FOOD
MOVE_RANDOM
IF_ CARRYING_FOOD
IF_MOVE_TO_ADJACENT_FOOD
IF_MOVE_TO_AWAY_PHEROMONE

The MOVE_RANDOM statement is for initially finding the food. It is not necessary to have the IF_FOOD_HERE operation. The ant program can execute a PICK_UP_FOOD instead, which will pick up the food if it is there, and do nothing if there is no food under the ant.

The solution to the problem would be:

```
IF_CARRYING_FOOD
        MOVE_TO_NEST
ELSE
        IF_MOVE_TO_ADJACENT_FOOD
                PICK_UP_FOOD
        ELSE
                IF_MOVE_TO_AWAY_PHEROMONE
                        PICK_UP_FOOD
                ELSE
                        MOVE_RANDOM
                        PICK_UP_FOOD
```

This is the simplest solution. Simply put, if the ant has food it keeps moving to the nest, until it gets there, and then automatically drops it. If there is food next to the ant, it picks it up. Otherwise, if there is pheromone scent away from the nest the ant moves to it, and then tries to pick up food. Otherwise, the ant moves randomly and then tries to pick up food.

The last two PICK_UP_FOOD statements show why there is no need for a IF_FOOD_HERE statement. The ant tries to pick up food any time it moves to a new location.

This problem could be a single ant problem, since only one ant is required to find all of the food. If the dissipation of the pheromones was slow enough, it could even use pheromones to find its way back to the pile of food. With multiple ants, the search for food initially, and the moving of food back to the nest would be much faster.

7.3 The Water Crossing Problem

The Water Crossing Problem is the first of two Multiple Ant Problems. Because it is a multiple ant problem, it will require at least 2 ants to solve..

The Water Crossing Problem requires the ants to build a bridge to cross a stream. Some of the ants must move into the water and die, to build a bridge for the other ants. The difficulty in this problem is having just some of the ants move in to the water and die, and not all or none of them. This needs to be done without the use of states or global control. The ants must all execute the same program with the same parameters, without variables or states that could cause some of the ants to act differently from others.

What makes this possible is that ants can tell what is directly next to them. This is not a state, because this information is lost as soon as the ant moves, but it allows the ant to act on an environmental condition. The ants need to be able act on information that will allow some to enter the water, while guaranteeing that some of them will not. They need to be able to detect what previous ants have done, or are doing now.

The following statements allow the ants to act based on these environmental conditions:

IF_MOVE_INTO_WATER
IF_MOVE_TO_DEAD_ANT
IF_MOVE_TO_AWAY_PHEROMONE
IF_ANOTHER_ANT_HERE

The IF_MOVE_INTO_WATER statement allow the ants to execute other statements before moving forward into the water and dying. This is besides the dropping of food which is done automatically. The ant could release a pheromone scent, to attract other ants to the bridge that it started to build. could execute the RELEASE_PHEROMONE statement several times, to create a very strong scent.

The IF_MOVE_TO_AWAY_PHEROMONE statement allows other ants to be attracted to this scent and find the bridge that the ant started.

The IF_MOVE_TO_DEAD_ANT statement allows ants to easily find and use the ant bridge built by other ants, once the pheromones gets the ant close to the bridge.

The IF_ANOTHER_ANT_HERE is another method of ensuring that not all the ants will kill themselves in the water. If the ant does not move into the water unless there as another ant at the same position, then at least 1 ant will not move into the water. It also means that at least 1 ant will be at the ant bridge after it is made.

The size of the water that the ants must cross is very important. If the stream is 1 block wide, then only 1 ant is required to build a bridge across it. That ant can release pheromones before it dies, to attract other ants to the bridge.

If the stream is wider than 1 block, then other ants must be attracted to the bridge that is partly built. The ants must also build the bridge in the same direction. That is why the IF_MOVE_INTO_WATER only allows the ant to move forward into the water.

Usually, once the ants find the food, they will create a new bridge when bringing the food back to the nest. The MOVE_TO_NEST_THROUGH_WATER statement moves the ant in the direction of the nest, moving into the water if it blocks the path. Once the new bridge is built, then the ants can use the pheromone trail as in the Single Ant Problem to find the food.

The number of ants required to solve this problem is X+1 where X is the distance across the stream. If the ants build a new bridge to go from the food to the nest, then the number of ants required becomes 2X+1, but many more ants will probably be required to solve this problem in a finite amount of time.

If the number of ants is greater than the number of water blocks in the environment, then the solution is trivial. Eventually, the entire water area will be converted into a large bridge, and there will still be ants left to solve what has become a Single Ant Problem.

7.4 The Heavy Food Problem

The Heavy Food Problem is the second Multiple Ant Problem. In this problem, several ants are required to lift one piece of food. The number of ants needed to lift a piece of food is equal to the weight of the food. Also in this problem, the food is spread out, instead

of being in piles. The ants must find a piece of food, and wait at that food until enough other ants show up that can lift up the food. Then all of the ants bring the food back to the nest together.

If the heaviest piece of food is of weight H, then at least H ants are required to solve the problem. If the sum of all of the weights of food is S then S ants can easily solve the problem with out any special instructions. If there are N pieces of food, then the problem becomes interesting when the number of ants is A is:

$$H \leq A \leq S - N$$

When there are less than S-N ants, then it is possible that all of the ants will be waiting at different pieces of food, and there will not be enough ants at any of the pieces of food to lift them. Because of this scenario, it is necessary for the ants to be able to give up and leave a piece of food, to help other ants lift up their food.

The special statements included for this scenario are:

IF_CANT_LIFT_FOOD
IF_TIRED_OF_WAITING
ESCAPE_PHEROMONE

The IF_CANT_LIFT_FOOD statement is used by the ants to determine if there are enough ants at the food to lift it. If the ants cannot lift the food, then they could release pheromones to attract other ants, or leave the food to find another piece of food that they can lift.

The IF_TIRED_OF_WAITING statement is allows the ants to leave a piece of food that they cant pick up at a random time. If the ant carries no food, there is food at the same location as the ant, then there is a 1 in *WAIT_ODDS* chance that the statement will be true.

The ESCAPE_PHEROMONE statement causes an ant to move forward until there are no pheromones next to the ant, or until the ant hits the edge of the map or an obstacle. This statement is useful in combination with the IF_TIRED_OF_WAITING statement. Otherwise, no matter what is executed in the IF_TIRED_OF_WAITING clause, if pheromones are being used, then the ant will go right back to the same food.

There are some special rules for multiple ants moving with one piece of food. First, the food cannot start moving until the start of the next iteration. Say that there are three ants lifting the food: 1, 2, & 3. The second ant is the last one to get to the food. Now 1 and 2 have already executed the ant program for the turn, but 3 has not. Even though there are enough ants to lift the food, 3 cannot start moving with the food, because 1 and 2 will not be able to follow.

On the start of the next turn, once the ants are able to lift the heavy piece of food, then one ant is declared the leader. This done automatically by the genetic program intepreter. The leader will be the lowest numbered ant trying to lift the food (the first one to execute the ant program). The leader will execute the ant program normally. There is a

problem for the rest of the ants, because all of the ants should theoretically be executing the ant program at the same time, and the environment should be the same for each ants execution. This is not true in the intepreter. To make up for this, it is assumed that if the environment was the same for two ants starting at the same location, then they would execute the same statements, and do exactly the same actions. This is a valid assumption. Therefore, after the leader has executed the ant program, all of the other ants carrying the same piece of food do not need to execute the ant program. They can automatically be moved to the same location as the leader ant. If the leader ant has dropped the food, then all of the ants will drop the food. If the leader ant has died in water, then all of the ants are moved to the location of the dead ant. For the next turn, there will not be enough ants to lift the food.

8 The Ant Genetic Program

8.1 Program Structure

The ant programs created by the genetic program are binary trees whose nodes are made of two types of statements: Leaf Statements and NonLeaf Statements. Leaf Statements are actions: MOVE_FORWARD, TURN_RIGHT, DROP_FOOD... NonLeaf Statements are conditions: IF_FOOD_AHEAD, IF_AT_NEST... If the condition is true, then the left child of the condition is executed. If the statement is false, then the right child is executed. Sometimes conditions have side affects: IF_MOVE_TO_ADJACENT_FOOD... In this statement, if there is adjacent food, then the and moves to it, and then executes the left child. Otherwise, the ant doesn't move, and the right child is executed.

8.2 Statements

The following is a list of Leaf Statements used by the genetic program.

Leaf Statement	Description
MOVE_RANDOM	Move the ant 2 spaces in any of the 4 cardinal directions.
MOVE_QUASI_RANDOM	Same as MOVE_RANDOM, but the random is weighted to most likely move the and forward, and least likely backward.
MOVE_TO_NEST	Move the ant 1 space toward the nest in a cardinal direction. It is assumed that ants know how to return to their nest without help.
MOVE_TO_NEST_THROUGH_WATER	Same as MOVE_TO_NEST, but if there is water in the way, move into it and die. If carrying the food, drop it where the ant dies.
TURN_LEFT	Turn Left.
TURN_RIGHT	Turn Right.
MOVE_FORWARD	Move one space forward.
PICK_UP_FOOD	Pick up food if at current location, and no food currently carried.
DROP_FOOD	Drop food if carried, and there is no food at this location, unless at nest.
RELEASE_PHEROMONE	Release pheromone smell.
MOVE_INTO_WATER	If water ahead, move into it and die. Drop food if carried.
ESCAPE_PHEROMONE	Move forward until no pheromone smelled, or until move a maximum distance.

Leaf Statement	Description
NOOP	Do nothing.

The following is a list of Leaf Statements used by the genetic program. All conditions execute the left child if the condition is true, and the right child if it is false. Side affects are in parenthesis.

Non-Leaf Statements	Description
IF_FOOD_AHEAD	If there is food in the space directly in front of the ant.
IF_FOOD_LEFT	If there is food in the space left of the ant.
IF_FOOD_RIGHT	If there is food in the space right of the ant.
IF_FOOD_ADJACENT	If there is food in any of the 4 spaces next to the ant.
IF_FOOD_HERE	If there is food at the same position as the ant.
IF_CARRYING_FOOD	If the ant is carrying food.
IF_AT_NEST	If the ant is at the nest location.
IF_MOVE_TO_ADJACENT_FOOD	If there is an adjacent piece of food (move to the food).
IF_MOVE_TO_AWAY_PHEROMONE	If there is a pheromone in one of the two directions away from the nest (move to the pheromone. move to the strongest if both directions have pheromones).
IF_MOVE_TO_DEAD_ANT	if there is a dead ant ahead (move to the dead ant).
IF_MOVE_TO_ADJACENT_PHEROMONE	if there is an adjacent pheromone (move to the strongest adjacent pheromone).
IF_MOVE_TO_NEST	if there is no obstacle blocking the way to the nest (move to the nest).
IF_MOVE_TO_NEST_THROUGH_WATER	if there is no obstacle blocking the way to the nest (move to the nest. If water ahead, move into the water and die and drop food).
IF_MOVE_INTO_WATER	If there is water ahead move into it and die. Drop food if carried, after executing clause.
IF_WATER_AHEAD	If there is water ahead.
IF_OBSTACLE_AHEAD	If there is a wall ahead.
IF_ANOTHER_ANT_HERE	If there is another living ant at this location.

Non-Leaf Statements	Description
IF_CANT_LIFT_FOOD	If not enough ants at this location helping to lift heavy piece of food.
IF_TIRED_OF_WAITING	If have been waiting for other ants for help to lift a heavy piece of food for a given period of time

The following is a list of items that may appear in the text output of the ant program, but are not actually statements that appear in the ant program.

Other Statements	Description
ELSE	Used to separate the if and else clause.
TAB	Tabs are used in the text output of the ant program instead of { } to identify child-parent relationships. Consecutive statements at the same tab are have the same parent. If the second statement has 1 extra tab, then it is a child of the first.
START	The { symbol is used internally by the complier to identify child-parent relationships. A pre-processor removes the tabs from the ant program and replaces them with { }, before passing the ant program to the compiler
END	The } symbol used by the complier.

8.3 Environment

The environment that the ants execute the ant program in contains several types of items:

Item	Description
Ants	Ants can interact with and help each other in the environment.
Nest	Where the ants start, and where they must bring the food back to.
Food	The food the ants must collect. The weight of the food is the number of ants required to lift it.
Water	If water separates the Food and Ants, then the ants must cross the water.
Dead Ant Bridge	Ants that move into the water die, and create a bridge that allows other Ants to cross the water.
Obstacle	A wall that ants cannot cross.
Pheromone	A smell that ants can create to mark locations, and attract other ants.

8.4 Initial Population

The genetic program can start with a completely random set of ant programs, a set of ant programs from a previous run, or a combination of both. When using the combination, the old ant programs and the new ant programs do not need to share the same instruction set.

Ant programs from previous runs are stored in ascii files. The name of the file contains how many crossovers had executed, and the run number. A master data file for the run contains the parameters and instruction set of the run. A subset of the saved ant programs can be used, or the entire file of ant programs can be added to the initial population.

Random ant programs are created with a combination of Leaf Statements and Non-Leaf Statements. The root tree node of the ant program is always a Non-Leaf Statement (so that the ant program is always at least 3 statements long). Two parameters (*LEAF_ODDS* & *NON_LEAF_ODDS*) determine if the children are leaf or non-leaf nodes. The chance that a child is a leaf node is:

LEAF_ODDS / (*LEAF_ODDS* + *NON_LEAF_ODDS*)

The odds that a child is not a leaf node is:

NON_LEAF_ODDS / (*LEAF_ODDS* + *NON_LEAF_ODDS*)

Once the type of the child node is determined, all statements of that type that are being used have an equal change of being selected. The higher the value of *NON_LEAF_ODDS* compared with *LEAF_ODDS*, the larger the initial ant programs will be. If *NON_LEAF_ODDS* is too large. The initial ant programs will be infinitely big, and the genetic program will run out of memory.

8.5 Fitness

For this genetic program, the lower fitness, the better the ant program. The fitness of the ant programs are determined by executing the ant programs in the given environment. When the ant program is completed the following variables are determined: the amount of food left, the amount of food not found, the number of turns the ant program was executed, and the number of nodes in the ant program tree. These variables are used in the following function to determine fitness:

fitness = foodLeft * *FOOD_LEFT_WEIGHT* +

foodNotFound * *FOOD_UNFOUND_WEIGHT* +

numTurns * *TIME_WEIGHT* +

(numNodes / *NODE_GROUP*) * *NODE_WEIGHT*

The NODE_GROUP - NODE_WEIGHT constants are used to keep the ant programs from getting too big, without penalizing an ant program for being one node larger than another.

Example values would be:

FOOD_LEFT_WEIGHT	= 10000000
FOOD_UNFOUND_WEIGHT	= 10000
TIME_WEIGHT	= 1
NODE_WEIGHT	= 10000
NODE_GROUP	= 50

With these values, the amount of food not returned to the nest is most important. The amount of food never found is next important. The size of the ant program is third, and the amount of time required to complete the ant program, or the fact that the ant program failed to find all the food in the given time, is least important.

This NODE_GROUP value allows all ant programs with 1 - 50 nodes to be given the same fitness. Only those with more than 50 nodes will get penalized for being too big. This should keep all ant programs less than 100 nodes, with most being under 50 nodes.

8.6 Tournament Selection

A tournament selection method is used for determining which ant programs to crossover. Every iteration *TOURNAMENT_SIZE* ant programs are selected from the population. The two ant programs in the tournament with the best fitness are used for crossover. The two new ant programs that are created replace the two worst ant programs from the tournament in the population.

8.7 Crossover

Before the crossover is done, a crossover point must be selected in each ant program. This is done by traversing down the tree, picking either the right or left child randomly. At a given point, the genetic program will stop traversing the tree, and were is stops is the crossover point. When the genetic program stops is determined by a crossover odds variable. The crossover odds variable is

$$\text{crossoverOdds} = \log_2(\text{numNodes}) + 1$$

and is at least 3. Every time the genetic program needs to make a decision on the left or right child in searching for a crossover point, there is a 1 in crossoverOdds chance that it will stop at the current node. Including the root node.

Since the ant programs are binary trees, crossover is done by simply switching the two pointers to the crossover nodes.

8.8 Mutation

Mutation is done after the crossover is complete. There is a one in *MUTATE_ODDS* chance that each child will have one node changed. If the ant program is to be mutated, then the individual chance that a node will be mutated is

mutateOdds = log2(numNodes) + 1

and is at least 2. The selection of the node to mutate works in the same way as the selection of a crossover point. The tree is traversed downward, randomly selecting right or left children, with a 1 in mutateOdds chance of stopping at the current tree node.

When the node to mutate is selected, it has an equal chance of being mutated into any of the other instruction statements being used that are of the same type (leaf or non-leaf).

The new ant program is evaluated with the fitness function before and after the mutation is done. The genetic program allows for greedy mutation. If this option is being used, then there is a *PERCENT_GREEDY_MUTATE* in 100 chance that if the new child with the mutation has a worse fitness than without the mutation, then the mutation will be undone.

Mutation allows fresh genetic material into the genetic program. Without mutation, eventually every ant program in the population would be the same.

8.9 Execution

The ant programs need to be executed in order to evaluate their fitness.

Initially, there are *NUM_ANTS* ants, all starting at the center of the nest, pointing South. For each iteration, the ant program is executed for each ant in order once. Changes to the environment such as: ants moving, picking up food, dropping food, moving food and ants building bridges take affect immediately. Other changes such as: release of pheromones and spread of pheromones are done after all ants have moved for the iteration.

The ant programs are executed for each ant *MAX_TURNS* number of times, or until all the food has been moved to the nest.

9 Input to the Genetic Program

9.1 Parameters

The following is a list of the parameters for the genetic program, that determine how it will run. The parameters are supplied by a params file. When the genetic program is executed, the first parameter determines which params file to use. Example, if the genetic program is run with the statement "ant 12", then the params file named params12 will be used. The options that can be specified in the params file, and their default values, are listed below.

Name	Default	Description
POPULATION_SIZE	1000	This is the number of ant programs in the population.
TOURNAMENT_SIZE	4	The number of ant programs from the population selected for the tournament for crossover.
FOOD_LEFT_WEIGHT	10000000	How much each piece of food not returned to the nest affects the fitness of the ant program.
FOOD_UNFOUND_WEIGHT	100000	How much each piece of food not found affects the fitness of the ant program.
TIME_WEIGHT	1	How much the time taken to move all of the food to the nest affects the fitness of the ant program.
NODE_WEIGHT	10000	How much the size of the ant program affects the fitness of the ant program.
NODE_GROUP	50	Used to group together ant programs with similar sizes, so that *NODE_WEIGHT* has a less affect on fitness.
LEAF_ODDS	9	The chance that the next node will be a leaf node, when creating the initial population.
NON_LEAF_ODDS	6	The chance that the next node will be a non-leaf node, when creating the initial population.
MUTATE_ODDS	100	The odds that the new ant programs created by crossover will be mutated.
PERCENT_GREEDY_MUTATE	80	The odds that the mutation will be greedy.
MAX_TURNS	1000	The maximum number of turns the ants are given to find all of the food.
NUM_ANTS	15	The number of ants that run the ant program in the environment.

Name	Default	Description
GOAL_FITNESS	0	The fitness that must be achieved for the genetic program to end.
WAIT_ODDS	0	The chance that an ant waiting for help to pick up food will leave.
AUTO_FOOD_DROP	1	Boolean for automatically dropping food when the ant is over the nest.
PHEROMONE_STRENGTH	300	The amount of pheromone released on a release pheromone command.
PHEROMONE_SPREAD	50	The amount of pheromone that is spread to adjacent squares each iteration.
POP_SEED	0	The random number generator seed when creating the initial population.
INTERP_SEED	0	The random number generator seed when executing the ant program. If the number is 0, then the random number generator is not seeded. Each ant program will be evaluated with a different seed.
DEBUG	0	Boolean for debug printf's.
SHOW_TIME	1000	How often in iterations the number of iterations, and time is displayed to the screen.
DUMP	1000	How often in iterations a full population dump to file is done.
MAX_ITTERATIONS	5000	Maximum number of iterations the genetic program can execute
RESTORE_ITTERATION	0	The iteration from a previous run to restore from.
RESTORE_VERSION	0	The version from a previous run to restore from.
RESTORE_AMOUNT	0	The amount of ant programs from a previous run to restore.
INSTRUCTION_VERSION	1	The instruction version file to use.
MAP_VERSION	1	The map version file
VERSION	1	The version of this run

9.2 Instruction Set

The instruction set for the ant program is specified by a special file. This file is called instructionX where X is a number equal to the parameter *INSTRUCTION_VERSION*. In this file, all of the possible statements are listed, with a 0 or a 1 before them to tell if that instruction is used.

9.3 Map

The map file for the ant program is specified in the same way as the instruction file. The file is called mapX where X is a number equal to the parameter *MAP_VERSION*. The map file is a 40 x 40 character map of the environment in which the ants execute the ant programs. The dimensions of the map are specified as the defines MAPX and MAPY in map.h.

The characters that can appear in a map file are:

Character	Item	Description
N	Nest	Where ants start and bring back food to.
W	Water	Can be crossed by ants moving on to the water and building an ant bridge.
B	Obstacle	Cannot be crossed by ants
1-9	Food	The number is the amount of ants required to pick up the food.

9.4 Program

The genetic program can start with a population from a previous run. Three parameters must be specified for this: *RESTORE_ITTERATION*, *RESTORE_VERSION*, and *RESTORE_AMOUNT*. If these values are not all 0, then the genetic program will load a population from a previous run from a file. The name of the file is

testXX.save.YYYYYY

where XX is a number equal to *RESTORE_VERSION*, and YYYYYY is a 6 digit number equal to *RESTORE_ITTERATION*, with leading 0's if necessary. The file contains all the ant programs from the previous run, in character format, separated by &'s. The amount of ant programs that will be read from the dump file is equal to *RESTORE_AMOUNT*, or the number of ant programs in the file, or *POPULATION_SIZE*, whichever is less. The entire file does not have to be read in for the new population. If *RESTORE_AMOUNT* is less than *POPULATION_SIZE*, then the rest of the population is created randomly.

9.5 First Program

Before the initial population is created, the Genetic Program will look for a file named first. If this file exists, then the program contained in this file will be loaded as the first program in the population.

This can be used as a method of testing a solution with the Genetic Program. If the population size is 1, then the best solution will be the fitness of the first solution. This method is a faster way to test a solution compared with the display program. It does not contain any graphics processing, and can test a solution within a second. The display program can take over a minute to test a solution.

10 Output of the Genetic Program

10.1 New Best

The genetic program keeps the user up to date on the best result so far. After creating the initial population it prints to the screen the best fitness value. It also does this every time a ant program is found that has the best fitness seen so far.

An example of the genetic program displaying the best fitness is:

```
Iteration =   148, Time = Thu May 9 18:10:33, Fitness = 30301000,
                        Number of Nodes = 21, Number of Turns = 1000,
                        Food Left = 3, Food Not Found = 3
```

The genetic program also writes this information to a file. The file name is of the format

testXX.YYYYYY

where XX is the parameter *VERSION*, and YYYYYY is the iteration with leading 0's so that it is 6 digits.

In this file is a header, which is the same information that is printed to the screen, and also a character dump of the best ant program. An example file, named test11.000148, would be:

```
Iteration =   148, Time = Thu May 9 18:10:33, Fitness = 30301000,
                Number of Nodes = 21, Number of Turns = 1000,
                Food Left = 3, Food Not Found = 3
IF (CARRYING FOOD) THEN
        MOVE QUASI RANDOM
        MOVE TO NEST THROUGH WATER
ELSE
        IF (MOVE TO ADJACENT FOOD) THEN
                PICK UP FOOD
                IF (FOOD HERE) THEN
                        PICK UP FOOD
                ELSE
                        MOVE QUASI RANDOM
        ELSE
                MOVE QUASI RANDOM
```

10.2 Show Time

Every *SHOW_TIME* iterations, the genetic program will print to the screen the current time, and how many iterations have been completed, the best fitness, and the average fitness. This will give an indication of how long the genetic program will take to reach *MAX_ITTERATIONS*. Also, combined with the display of the new best data, it will show approximately how long it has been since a new best ant program has been found.

An example of the show time output is:

```
Iteration =   1000, Time = Thu May 9 18:23:12,
                  Average Fitness = 310914213, Best Fitness =  50501000
```

10.3 Program Dump

Every *DUMP* iterations, the entire population is dumped to a file. The name of the file is:

testXX.save.YYYYYY

where XX is a number equal to *VERSION*, and YYYYYY is a 6 digit number equal to the iteration, with leading 0's if necessary.

This file contains all of the ant programs, in the order that they exist in the population, not by fitness. The ant programs are separated by a line containing the character "&".

11 Interface

11.1 Input

Along with the genetic program is a graphical interface that allows the user to see the ants execute an ant program. The option input to this display is a file that contains aa ant program for the ants to run. This file must be in the same format as the files that are automatically dumped by the genetic program. To run the interface with the ant program dumped in the example from above:

display test11.000148

11.2 Configuration

The display program does not use a params file. Instead, the user can modify values of parameters using slide bars on the screen. This allows the user to run the display program under several different environments. The parameters the user is allowed to modify are: *MAP_VERSION*, *PHEROMONE_SPREAD*, *PHEROMONE_STRENGTH*, *MAX_TURNS*, and *NUM_ANTS*. There is also a slider for controlling how quickly the ants move, which affects the amount of time it takes the display program to execute an ant program.

Besides the sliders, there is a text field above the sliders, where the user can enter the name of an ant program to execute. If the user entered an ant program name on the command line, then that name will appear in the name field.

Below the sliders are the execute and quit buttons. The execute button is also the stop button, when the ant program is running.

To the right of the sliders is a display of the map where the ant program will execute. Below the file name field, and above the sliders, the variables that determine fitness are displayed: Fitness, Num Nodes, Num Turns, Food Left, and Food Not Found.

The map slider determines which map file will be used. The values of the slider range from 1 - 40, specifying map file map1 - map40. When the slider value is moved, the display area to the right of the sliders will change to display the currently selected map file. This slider can be changed while the ant program is running, and the map file will be changed in the middle of execution. This could be done to reset the positions of food.

The pheromone parameters determine how much pheromone is released, and how fast it spreads in the map. These parameters can also be changed during execution to increase or decrease the effect of pheromones. A pheromone spread value of 0 means that pheromones do not spread.

Max turns determines how many turns the ant program will run. When the ant program reaches this number is stops. The max number of turns can be increased after this, to get the ant program to continue, or decreased to make the execution stop. The current turn number is displayed on the screen, as part of the fitness information.

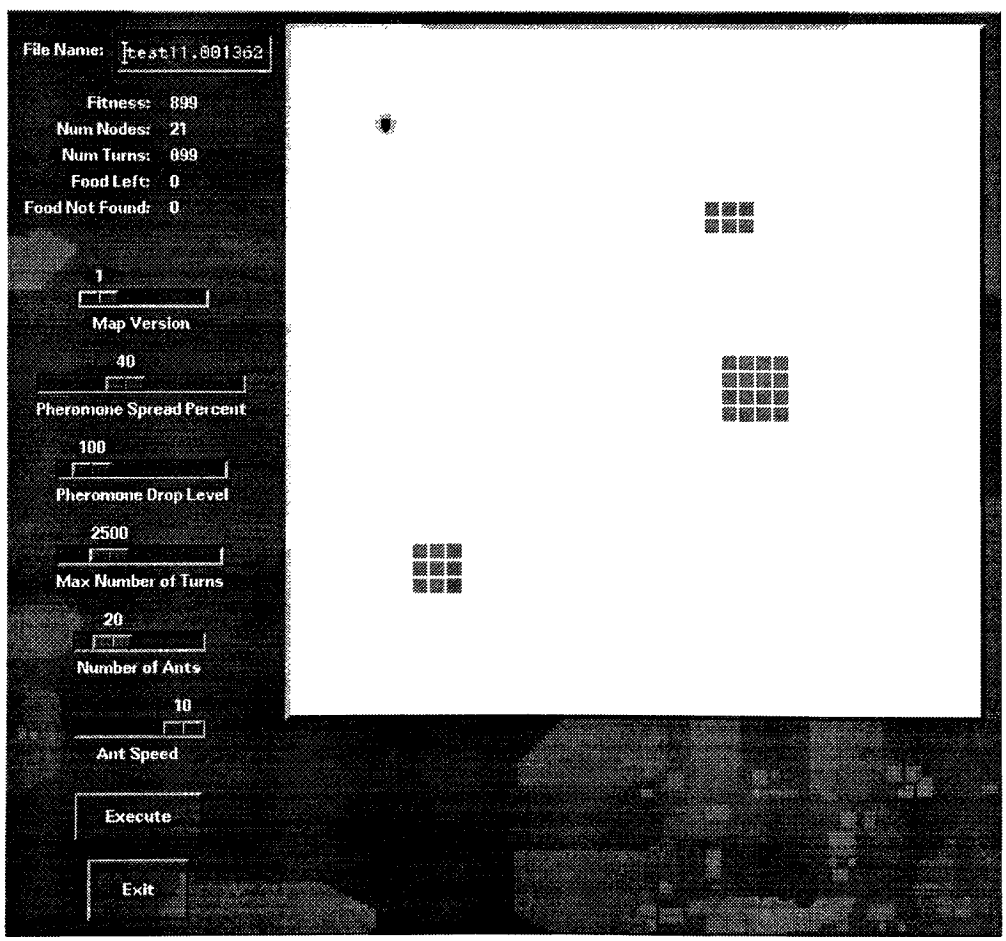
The number of ants slider determines how many ants will run the ant program, and appear on the map. If this number is changed, then ants will appear or disappear from the map. No special provisions are made for dead ants or ants carrying food.

The file name field shows the name of the currently loaded ant program. If the ant program is not a valid ant program file, then the label of the execute button will change to "Need File Name". When a legal file has been loaded, the execute button label will change to "Execute". When the ant program is running, the execute button label will change to "Halt".

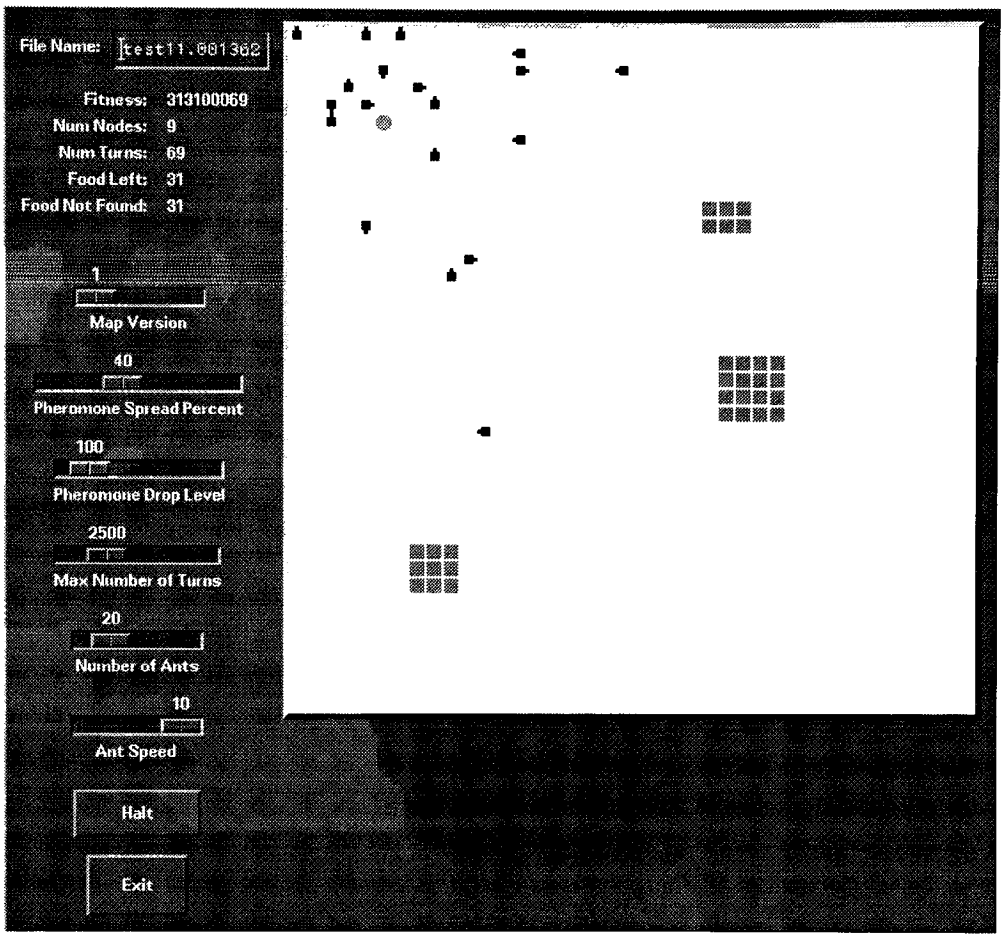
The fitness information is updated after each iteration (after each ant has executed the ant program once). The fitness value is determined using the different default: *XX_WEIGHT* parameter values.

11.3 Execution

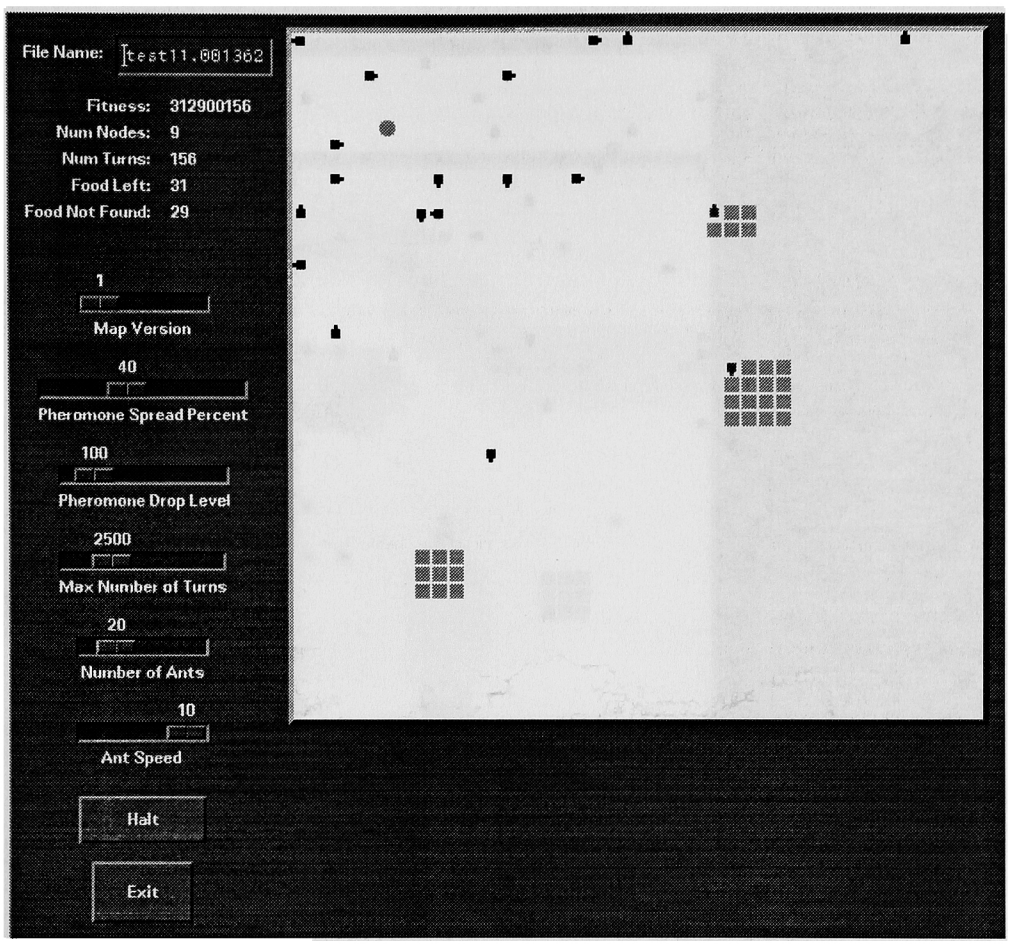
When a legal program file name has been entered in the file name text field, and a legal map file has been selected, then the execute button will become selectable. All of the ants will still be at the nest. The display will look something like this:



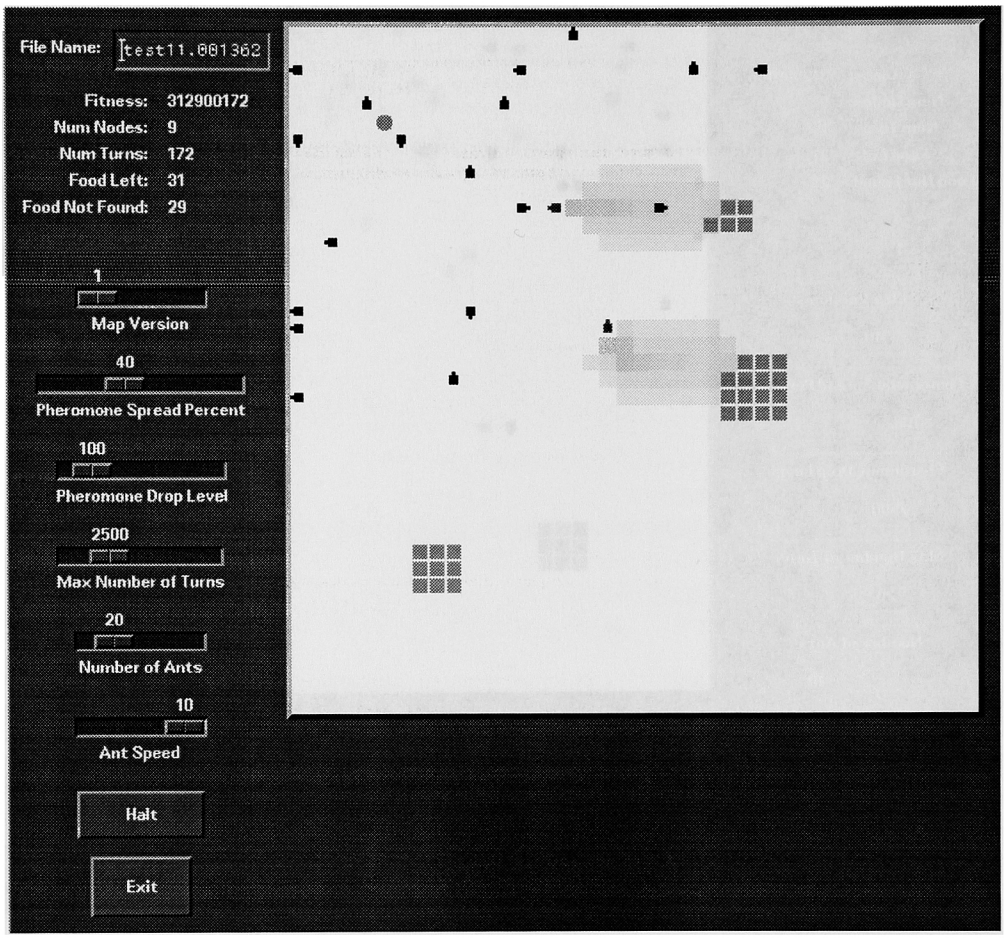
When the execute button is pressed, the ants will start moving. Before execution, all of the fitness values were from the original run by the genetic program. Now the fitness values represent those of this current display run. The Execute button has changed to a Halt button. The user will be able to see the ants search for food, as seen below:



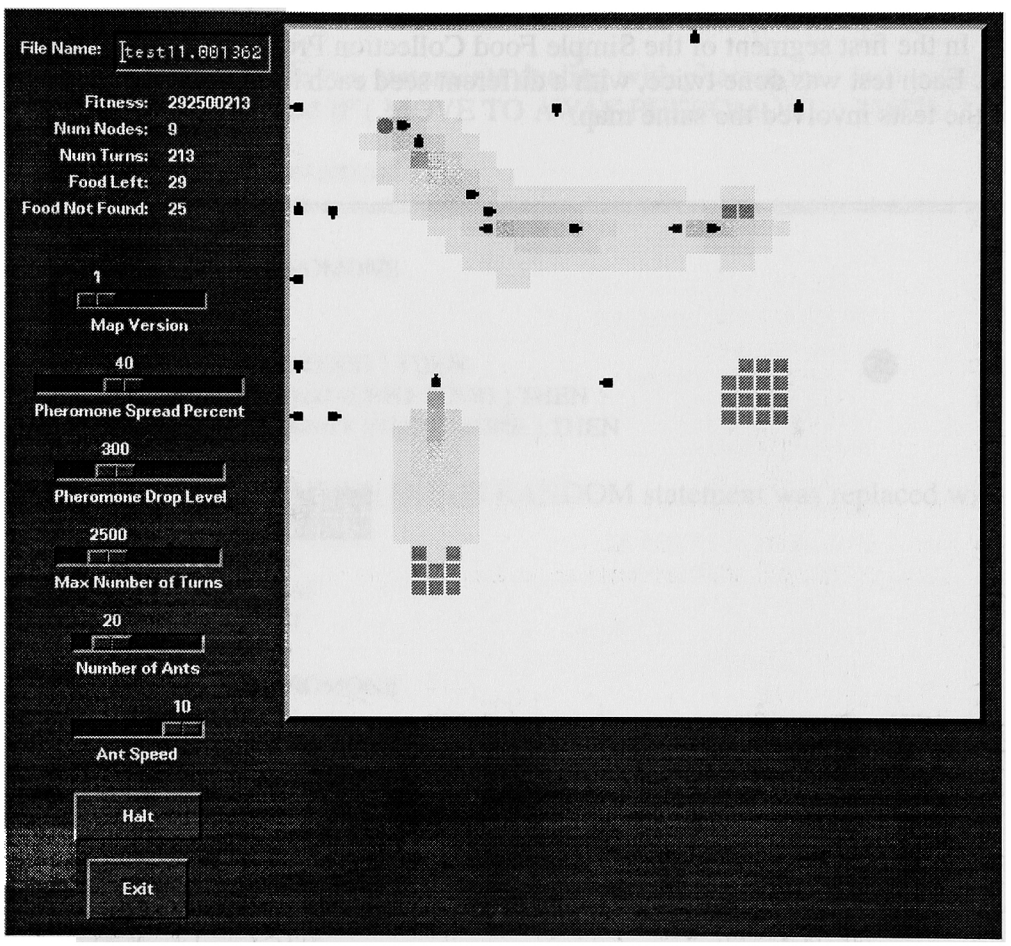
Here we can see that 2 ants have found two different piles of food at the same time. These 2 ants have picked up the food, and it no longer appears on the map. In black and white, the color change in the ant can barely be seen. In color, the ant would change from brown to green. The Food Not Found value has also been updated.



Now the two ants are moving back to the nest with the food, leaving a pheromone trail as they move to the nest. One other ant has already found the top pheromone trail, and is following it back to the food. The strength of the pheromone trail can be seen by the different shades of gray.



Here, the top pile of food has a pheromone trail leading to the nest. This pile of food is almost gone, and there are many ants on this trail. Some are bringing food back to the nest, and others are heading toward the food. The middle pheromone trail has completely dissipated. The Pheromone Drop Strength value was not large enough. Therefore, the value has been changed from 100 to 300, while the program was running. One ant has discovered the bottom pile of food, and is bringing food back to the nest. Hopefully, this trail will not dissipate, now that the pheromone strength is 3 times stronger. The Food Left value has now also changed.



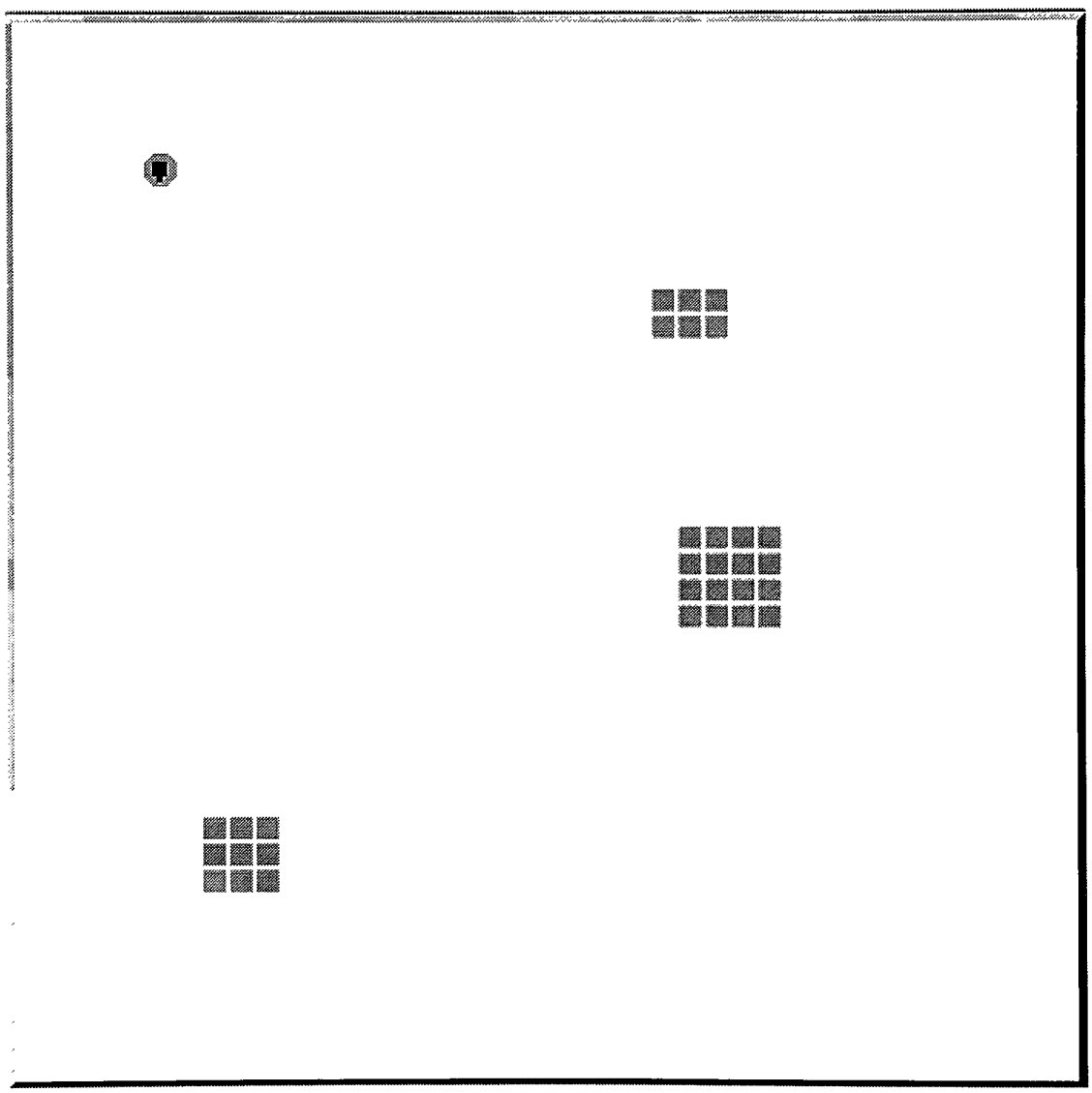
12 Results

12.1 Simple Food Collection Problem (Test 1-4)

12.1.1 Input Parameters

In this problem, the food is in groups. There are no obstacles or water in the map. Only one ant is required to lift a piece of food.

In the first segment of the Simple Food Collection Problem, four tests were conducted. Each test was done twice, with a different seed each time, for a total of eight tests. All of the tests involved the same map.



All four tests shared had basically the same instruction set. The instruction set for the first test was:

```
MOVE QUASI RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE

PROC2
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
```

For the second test, the two statements dealing with pheromones were removed: RELEASE PHEROMONE and IF (MOVE TO AWAY PHEROMONE) THEN.

```
MOVE QUASI RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE

PROC2
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
```

For the third test, the MOVE QUASI RANDOM statement was replaced with MOVE RANDOM.

```
MOVE RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE

PROC2
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
```

For the fourth test, the IF (FOOD HERE) THEN statement was added.

```
MOVE RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE

PROC2
IF ( FOOD HERE ) THEN
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
```

The parameters for the tests were the same, except for VERSION, INSTRUCTION VERSION, and the POP_SEED. The parameters were:

POPULATION_SIZE	500
TOURNAMENT_SIZE	4
FOOD_LEFT_WEIGHT	10000000
FOOD_UNFOUND_WEIGHT	100000
TIME_WEIGHT	1
NODE_WEIGHT	10000
NODE_GROUP	50
LEAF_ODDS	9
NON_LEAF_ODDS	6
MUTATE_ODDS	100
PERCENT_GREEDY_MUTATE	80
MAX_TURNS	1000
NUM_ANTS	15
GOAL_FITNESS	0
POP_SEED	0 or 1
INTERP_SEED	0
PHEROMONE_STRENGTH	300
PHEROMONE_SPREAD	50
WAIT_ODDS	0
AUTO_FOOD_DROP	1
DEBUG	0
SHOW_TIME	100
DUMP	10000
MAX_ITERATIONS	5000
RESTORE_ITERATION	0
RESTORE_VERSION	0
RESTORE_AMOUNT	0
INSTRUCTION_VERSION	1, 3, 5, or 7
MAP_VERSION	1
VERSION	1, 2, 3, 4, 5, 6, 7, or 8

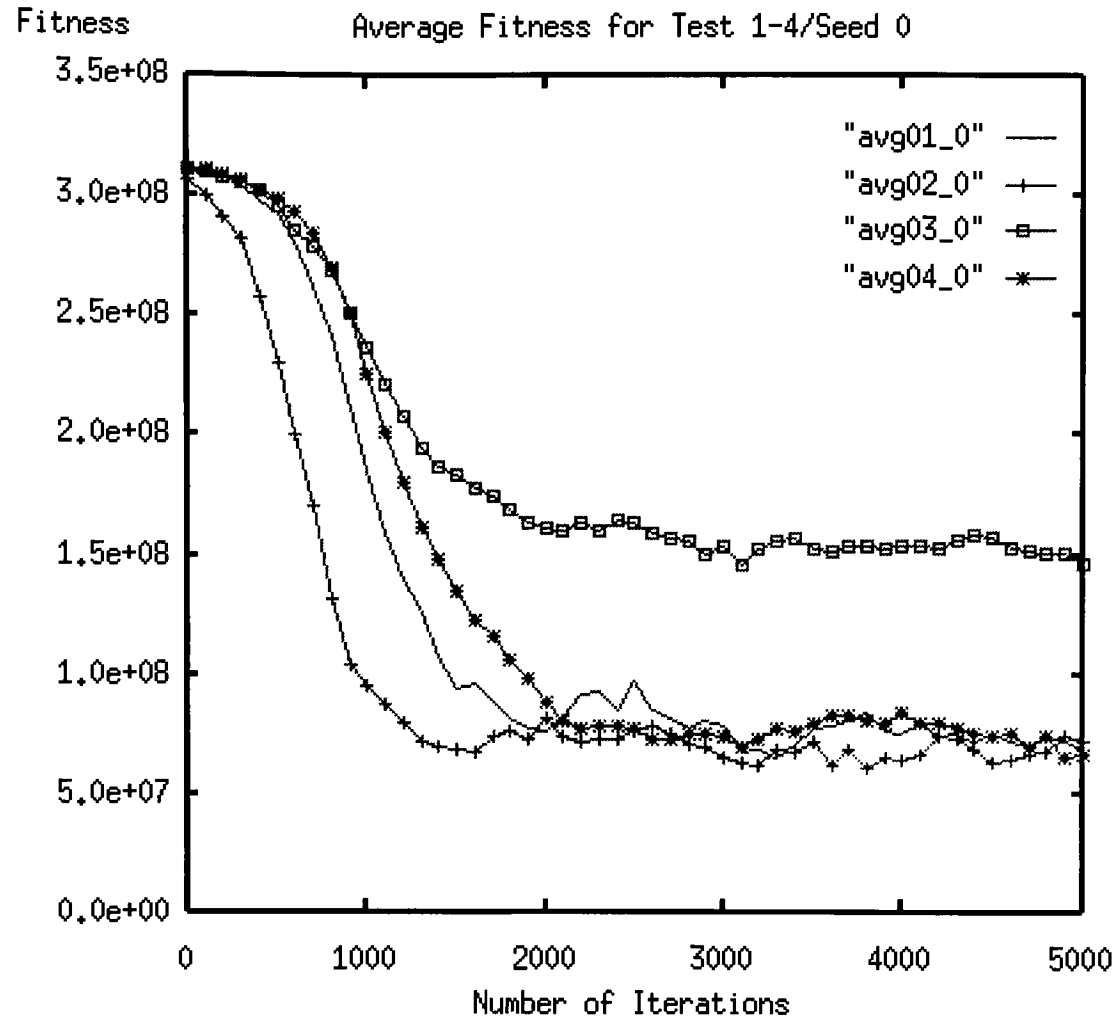
12.1.2 Output Data

The following four plots show the average and best fitness for both seeds.

Line 1: ---	Solid Line	Standard Instruction Set
Line 2: -+-	Solid Line with Crosses	Pheromone Instructions Removed
Line 3: -[]-	Solid Line with Squares	Move Random instead of Move Quasi Random
Line 4: -*-	Solid Line with Stars	Added If (Food Here) Then Statement

Analyzing the results of the tests will lead to the need of further testing. Any test that requires more testing for clarification will be mentioned in the analysis. More analysis will be done after the solutions produced by the Genetic Program are examined.

The numbers in the legend after avg or best refers to the test number and seed.



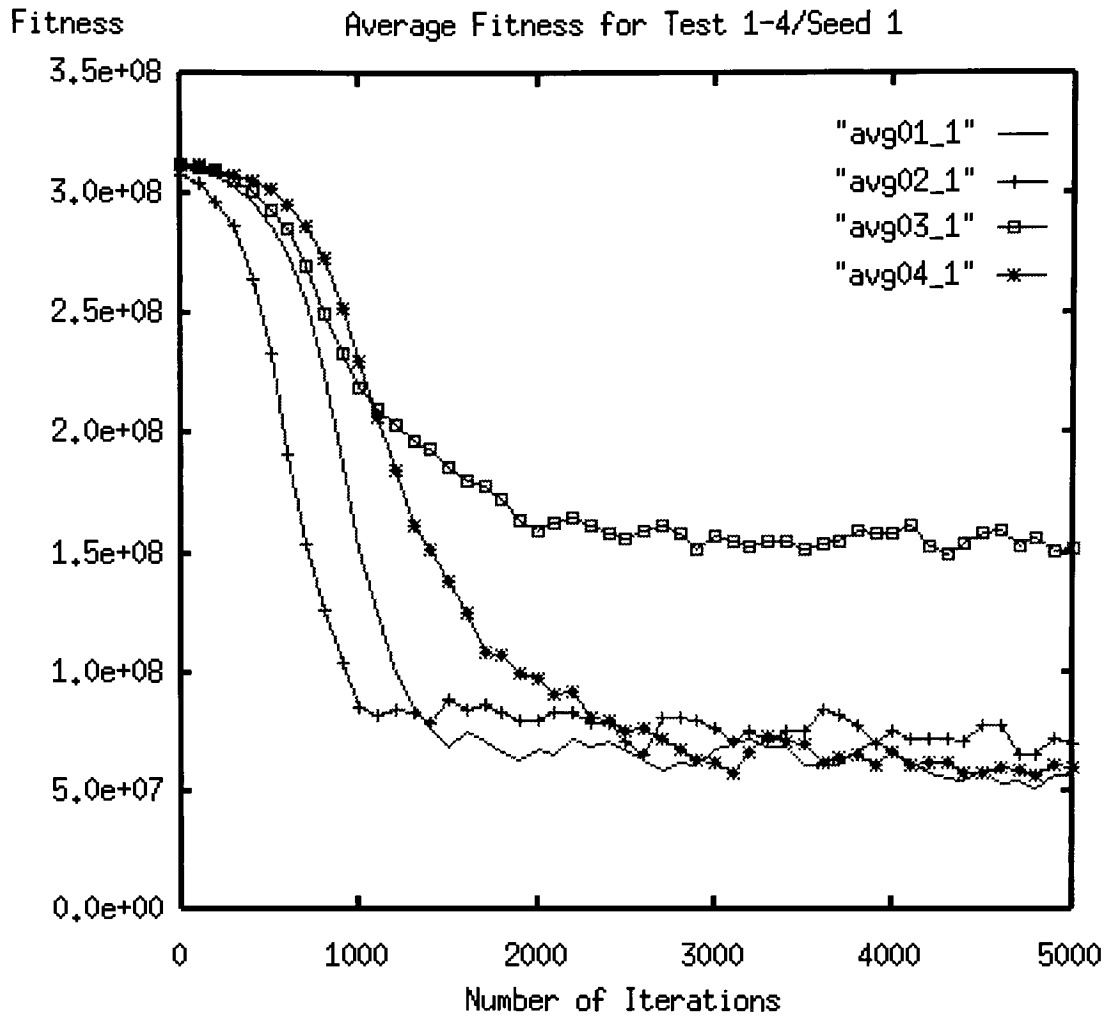
This graph shows the average fitness of the population. It shows that the test using the MOVE RANDOM statement did not work well compared with the other 3 tests. Remember, that MOVE QUASI RANOM statement has the following effect that the ant will move 2 squares in a random direction, but the random that the ant turns before moving is weighted:

50% chance ant will not turn moving

20% chance ant will turn either left or right

10% chance the ant will turn around

The MOVE QUASI random allows the ants to spread out much faster, allowing them to find food better. If the ants were allowed more time to find the food, the MOVE RANDOM would probably worked better, and allowed the ants to find all of the food. The only fitness difference would have been the number of turns to find the food. Another test will be required to prove this point.

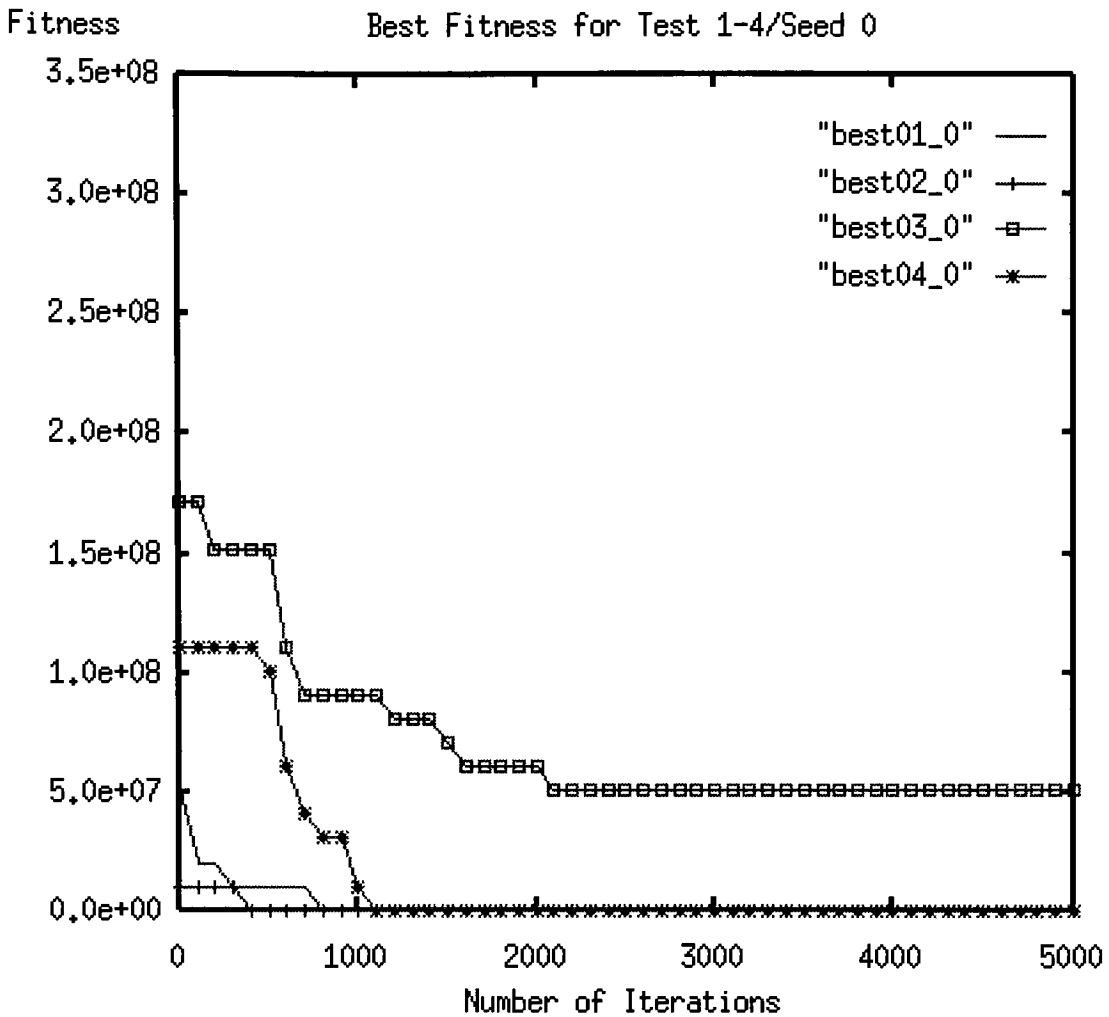


This graph is much like the first. The order of the tests is the same.

Best:	Pheromone Instructions Removed
	Standard Instruction Set
	Addition of IF FOOD HERE statement
Worst:	MOVE RANDOM instead of MOVE QUASI RANDOM

It appears that the IF FOOD HERE statement did not lead to a better solution. It only complicated the work of the Genetic Program by adding another statement. This just increased the size of the search space unnecessarily.

The removal of the Pheromone statements actually helped the Genetic Program find a solution faster. It appears that the Pheromone statements were unnecessary, and also only served to increase the size of the search space. This could happen for two reasons. First, the food was too easy to find, and pheromones were not required to relocate the pile of food.

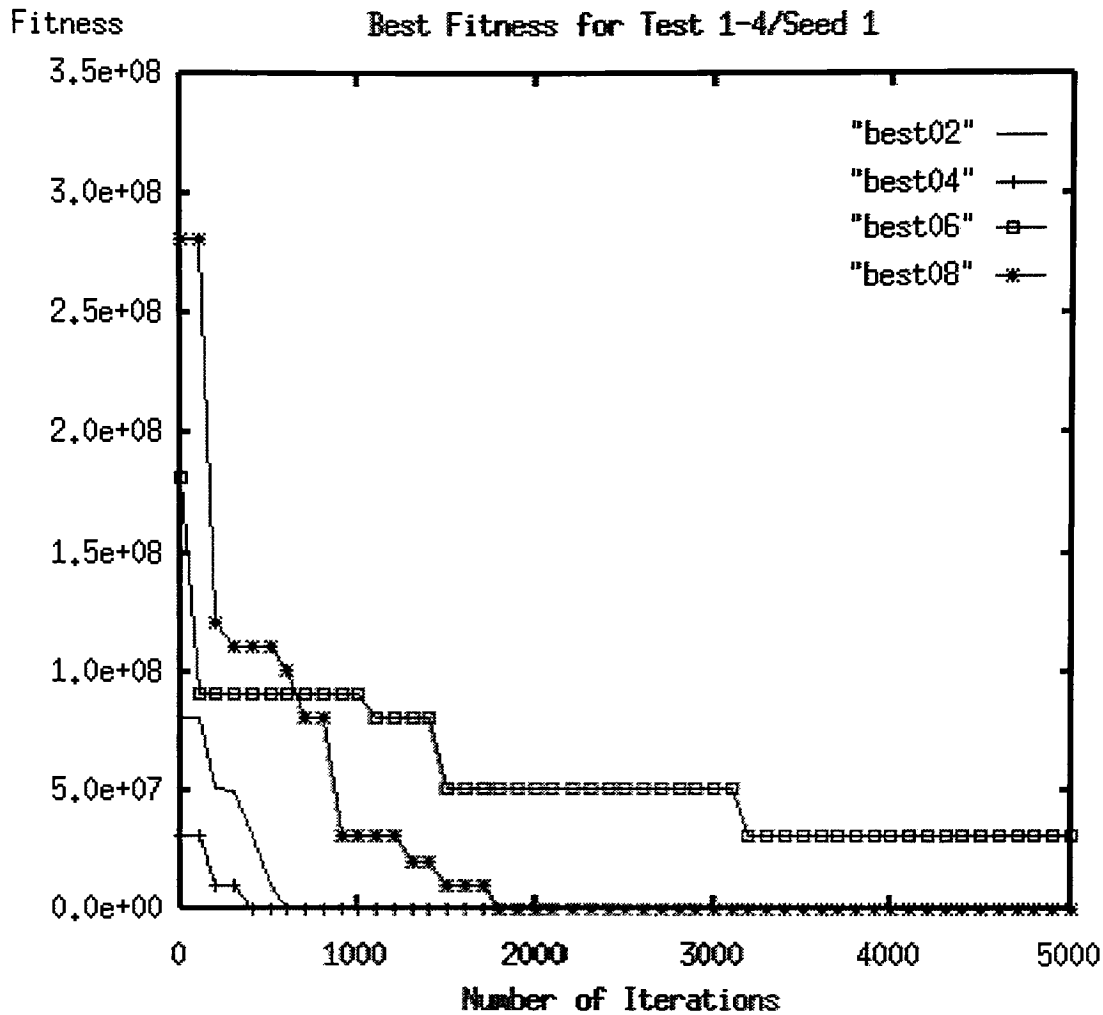


Second, the pheromones dissipated too quickly to be used by the other ants. This also requires further testing to prove. The display program will be able to show if the pheromones were dissipating too fast. More test will prove if pheromones can be useful under any configuration.

This third graph shows the best fitness of the population. The 4 tests rank in the same order here as they did in the average fitness graph. Looking at the best fitness of the initial population, it can be seen that the best odd for having a good solution in the initial set of solutions is to have a very simple instruction set.

From these graphs, it also appears that after 2000 iterations, not much is accomplished by the Genetic Program. Further related testing should not have to go beyond 2500 iterations.

The fourth graph on the next page also supports the findings of the other three graphs.



These are the final solutions found for each test. I have *italicized* the statements that have no meaning, or cannot be reached.

Test 1 - Standard Instruction Set - seed 0

```
Iteration =  2112, Time = Tue Feb 11 23:41:21, Fitness =     734,
     Number of Nodes =  9, Number of Turns =  734,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  IF ( CARRYING FOOD ) THEN
    MOVE QUASI RANDOM
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      MOVE QUASI RANDOM
      PICK UP FOOD
```

Test 1 - Standard Instruction Set - seed 1

```
Iteration =  4389, Time = Wed Feb 12 01:21:46, Fitness =     692,
     Number of Nodes =  31, Number of Turns =  692,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    MOVE QUASI RANDOM
IF ( CARRYING FOOD ) THEN
  IF ( CARRYING FOOD ) THEN
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
      MOVE TO NEST
    ELSE
      MOVE TO NEST
      PICK UP FOOD
  ELSE
    IF ( CARRYING FOOD ) THEN
      IF ( CARRYING FOOD ) THEN
        RELEASE PHEROMONE
      ELSE
        MOVE QUASI RANDOM
    ELSE
      RELEASE PHEROMONE
    IF ( CARRYING FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        RELEASE PHEROMONE
      ELSE
        IF ( MOVE TO AWAY PHEROMONE ) THEN
          PICK UP FOOD
        ELSE
          MOVE QUASI RANDOM
ELSE
  PICK UP FOOD
```

The first solution is straight forward. When the unused statements in *italic* are removed from the second solution, it becomes much like the first solution. The important thing that is discovered from these solutions is that the Genetic Program was not using Pheromone instructions to solve the problem. These are the solutions that would be expected from the second test.

This verifies why the second test found a solution faster than the first test. The Pheromone instructions were not used, and only served to increase the search space. A new test with stronger pheromones (greater *PHEROMONE_STRENGTH*), should prove whether pheromones are useless, or were dissipating too fast to be used.

Test 2 - Pheromone Instructions Removed - seed 0

```
Iteration =   962, Time = Tue Feb 11 21:56:57, Fitness =      755,
     Number of Nodes =  13, Number of Turns =  755,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  IF ( CARRYING FOOD ) THEN
    MOVE TO NEST
  ELSE
    MOVE QUASI RANDOM
ELSE
  PICK UP FOOD
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  MOVE QUASI RANDOM
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    PICK UP FOOD
```

Test 2 - Pheromone Instructions Removed - seed 1

```
Iteration =  4398, Time = Tue Feb 11 23:49:42, Fitness =      726,
     Number of Nodes =  13, Number of Turns =  726,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( CARRYING FOOD ) THEN
      MOVE TO NEST
      IF ( MOVE TO ADJACENT FOOD ) THEN
        MOVE TO NEST
      ELSE
        PICK UP FOOD
    ELSE
      PICK UP FOOD
  ELSE
    MOVE QUASI RANDOM
```

These solutions are again similar to the first test. Note that the first test's best solution in the first 5000 iterations was found at iteration 962, and the second test's best solution was found at iteration 4398. This does not mean that the second test took 4 times as long to find a solution. Looking at the graphs, tests 1 and 2 for both seed found solutions that found all of the food within the first 700 iterations. The Genetic Program will slowly find better solutions, because it finds it will find all of the food in less turns. The *INTERP _SEED* is not being set in these tests, so the same solution evaluated with two different seeds will have two different fitnesses. Thus, the best fitness for the Genetic Program will slowly get better, even though the best solution may not change.

Test 3 Move Random instead of Move Quasi Random - seed 0

```
Iteration =  2904, Time = Wed Feb 12 00:01:24, Fitness =  50401000,
     Number of Nodes =  35, Number of Turns = 1000,
     Food Left =  5, Food Not Found =  4
IF ( MOVE TO ADJACENT FOOD ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        IF ( CARRYING FOOD ) THEN
          PICK UP FOOD
        ELSE
          RELEASE PHEROMONE
      ELSE
        RELEASE PHEROMONE
    ELSE
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        PICK UP FOOD
      ELSE
        IF ( MOVE TO AWAY PHEROMONE ) THEN
          PICK UP FOOD
        ELSE
          IF ( MOVE TO AWAY PHEROMONE ) THEN
            MOVE RANDOM
          ELSE
            MOVE TO NEST
          IF ( MOVE TO AWAY PHEROMONE ) THEN
            IF ( MOVE TO ADJACENT FOOD ) THEN
              MOVE RANDOM
            ELSE
              PICK UP FOOD
          ELSE
            MOVE TO NEST
  MOVE RANDOM
ELSE
  IF ( CARRYING FOOD ) THEN
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      MOVE TO NEST
      RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      MOVE RANDOM
    ELSE
      MOVE RANDOM
  PICK UP FOOD
```

Test 3 - Move Random instead of Move Quasi Random - seed 1

```
Iteration =  3165, Time = Tue Feb 11 23:46:11, Fitness =  30301000,
     Number of Nodes =  13, Number of Turns = 1000,
     Food Left =  3, Food Not Found =  3
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
    ELSE
      PICK UP FOOD
      MOVE RANDOM
      MOVE RANDOM
  ELSE
    MOVE RANDOM
```

The statements in this solution are italicized, because with a RELEASE PHEROMONE statement in the program, the IF (MOVE TO AWAY PHEROMONE) THEN condition will never be true.

The best solution from test 3, seed 1 is also like the results of the first two tests. Yet, it did not find all of the food. This must be because it could not find the food fast enough, as hypothesized earlier. The MOVE RANDOM statement does not allow the ants to spread out fast enough.

Test 3 seed 0 does use pheromones. It releases pheromones on the way back to the nest with food, and ants following the pheromone trail release their own pheromones. Sometimes, the ants will follow the pheromone trail, but there is no direct schema linking the following of pheromones with looking for food. The fitness of this solution was not as good as seed 0. This is probably because the ants spent too much time incorrectly following pheromones.

Test 4 - Added If (Food Here) Then Statement - seed 0

```
Iteration =  4229, Time = Wed Feb 12 00:33:45, Fitness =     739,
     Number of Nodes =  31, Number of Turns =  739,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  IF ( MOVE TO AWAY PHEROMONE ) THEN
    PICK UP FOOD
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      RELEASE PHEROMONE
  ELSE
    MOVE TO NEST
ELSE
  IF ( MOVE TO AWAY PHEROMONE ) THEN
    MOVE QUASI RANDOM
  ELSE
    PICK UP FOOD
MOVE QUASI RANDOM
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      IF ( FOOD HERE ) THEN
        RELEASE PHEROMONE
        IF ( FOOD HERE ) THEN
          IF ( FOOD HERE ) THEN
            MOVE QUASI RANDOM
          ELSE
            MOVE TO NEST
        ELSE
          MOVE TO NEST
      ELSE
        PICK UP FOOD
    ELSE
      MOVE TO NEST
  ELSE
    PICK UP FOOD
```

Test 4 - Added If (Food Here) Then Statement - seed 1

```
Iteration =  4012, Time = Wed Feb 12 04:40:18, Fitness =     703,
     Number of Nodes =  23, Number of Turns =  703,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  PICK UP FOOD
IF ( CARRYING FOOD ) THEN
  PICK UP FOOD
ELSE
  MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    PICK UP FOOD
  MOVE TO NEST
  MOVE QUASI RANDOM
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE TO NEST
  ELSE
    PICK UP FOOD
PICK UP FOOD
PICK UP FOOD
```

The second seed for this test is almost a perfect solution. Although there are many extra statements, the solution is basically move random until food is found, then move to nest. The difference here is that the move to nest is a drunken walk. The ant does a MOVE TO NEST, and then a MOVE QUASI RANDOM.

The solution for the first seed uses pheromones. The ant will only release a pheromone if there is food at its location. It doesn't release pheromones on the way to the nest. The ants do try to follow the pheromones, but they probably dissipate too fast to be used. It does contain the IF (MOVE TO ADJACENT FOOD) THEN PICK UP FOOD and the IF (CARRYING FOOD) THEN MOVE TO NEST schemata, and most of the other statements will rarely be used.

As for the IF (FOOD HERE) condition, it is only used by the seed 0 test to release pheromones when there are pheromones and food at the ants location. For the most part, the addition of the IF (FOOD HERE) condition only served to complicate the work of the Genetic Program by increasing the search space.

12.1.3 Synopsis

From these tests for the Simple Food Collection Problem, it was learned that:

1) The MOVE QUASI RANDOM statement is much better than the MOVE RANDOM statement in getting the ants to find food.

2) The pheromone parameters that were used: *PHEROMONE_STRENGTH* = 300 and *PHEROMONE_SPREAD* = 50, allowed the pheromones to dissipate too fast. If the pheromones spread less, then they would last longer, but the pheromone trail would be thin. If the strength was increased, then the trail would take longer to dissipate, and the trail would be wide.

3) The IF (FOOD HERE) THEN statement is unnecessary, and complicates the work of the Genetic Program by increasing the search space.

4) For all the tests, for the given population size of 500, the best result was found within 2500 iterations. At this point, either the best solution was found, or the Genetic Program ran out of genetic material, and all of the solutions in the population were similar.

The best program for the Simple Food Collection Problem without use of pheromones appears to be:

```
IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
ELSE
        IF ( MOVE TO ADJACENT FOOD ) THEN
                PICK UP FOOD
        ELSE
                MOVE QUASI RANDOM
```

12.2 Simple Food Collection Problem (Test 5-6)

12.2.1 Input Parameters

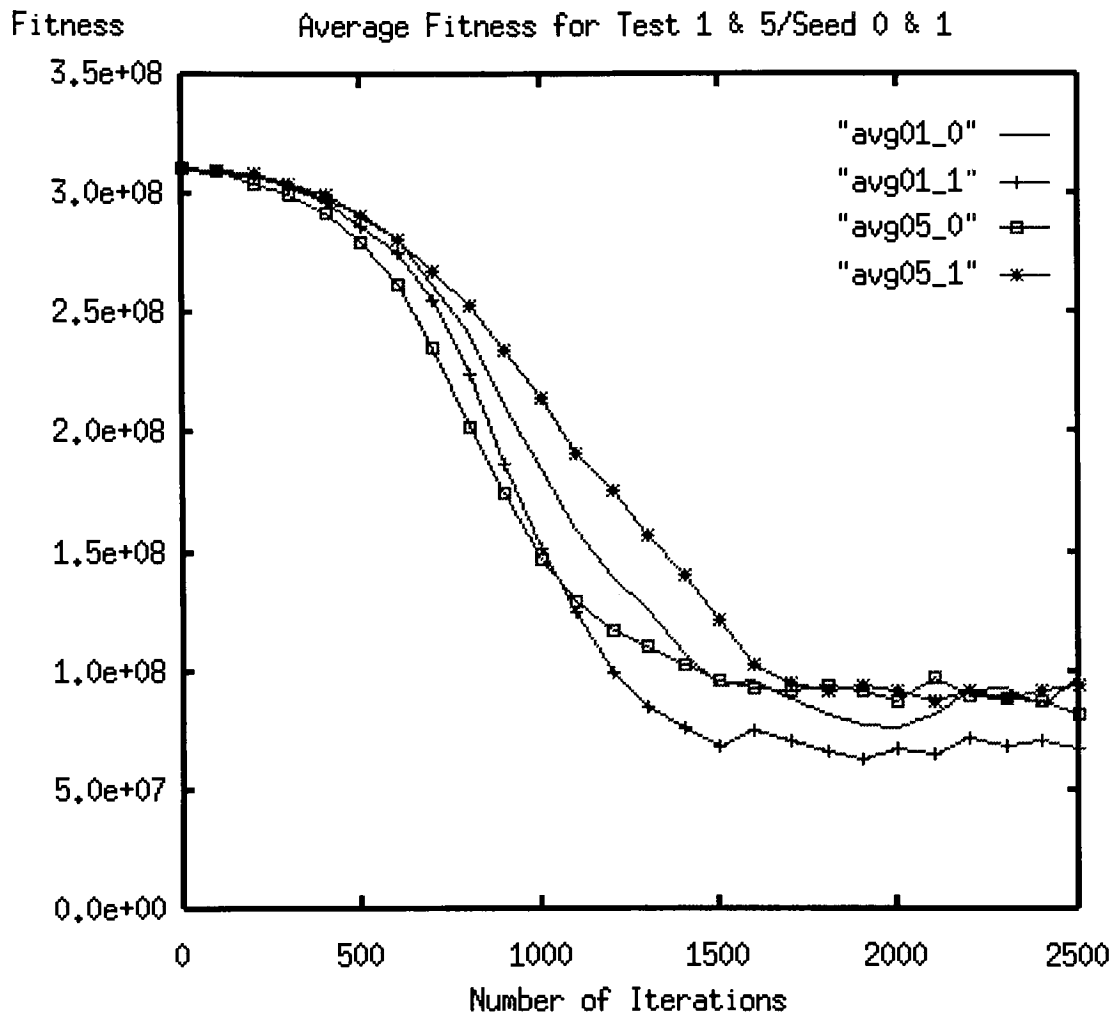
Two more tests (5 and 6) were run with the same map file. Test 5 used the instruction set from the first (standard) test for the Simple Food Collection Problem, and basically the same parameters except for *PHEROMONE_STRENGTH*. This test was used to prove that the ants could use pheromones advantageously, if the strength of the pheromone was stronger. Test 6 used a new instruction set that is a combination of the second (pheromones removed) test and the third (use MOVE RANDOM) test for the Simple Food Collection Problem. It also used basically the same parameters except for *MAX_TURNS*. This test was done to prove that the MOVE RANDOM statement could be used instead of MOVE QUASI RANDOM, if the max number of turns the ants were given to find all of the food was increased. These tests (5 and 6) were each run with 2 different seeds. The *MAX_ITERATIONS* for both tests was dropped to 2500. Any more is unnecessary.

Here are the parameters of all the tests, with only the changes shown for tests 5 and 6.

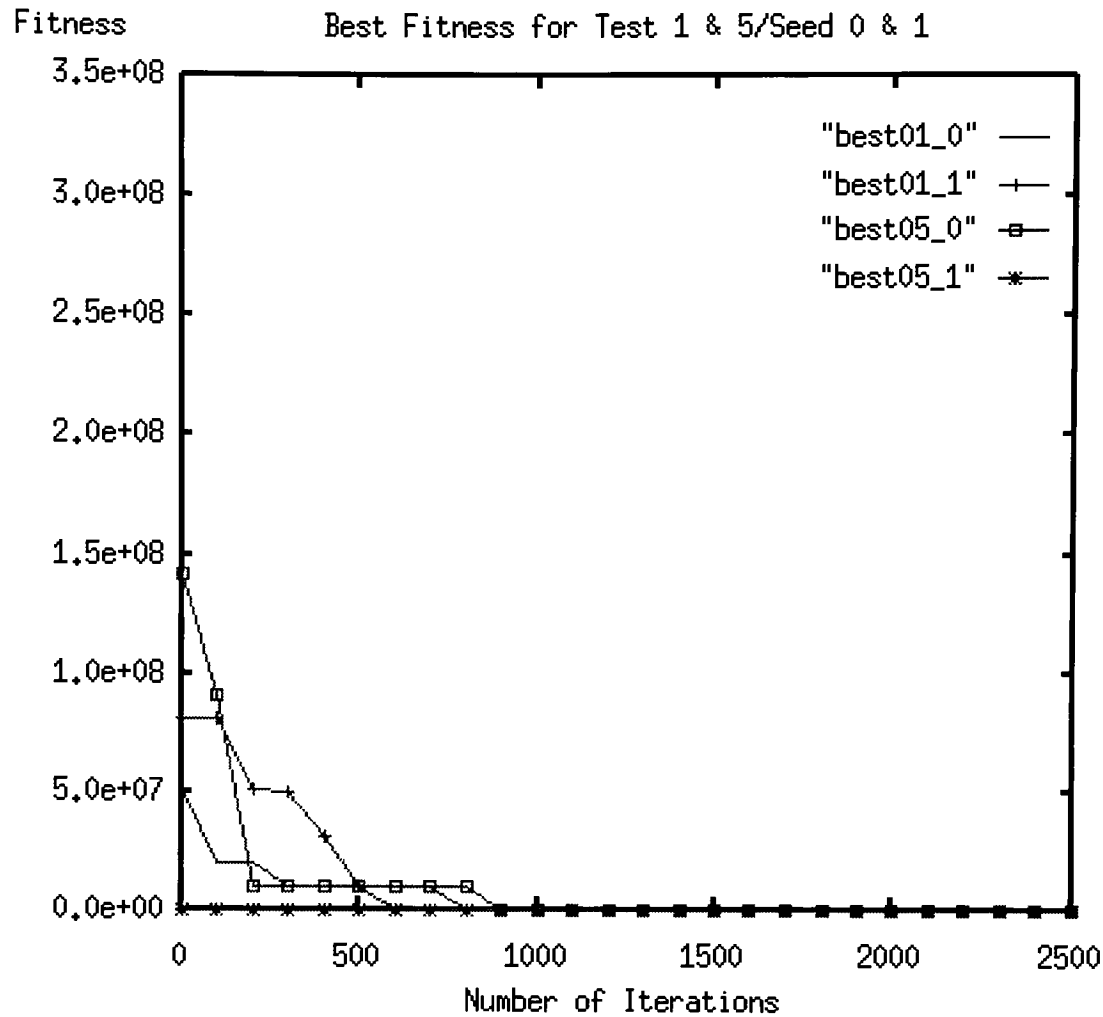
Parameter	Tests 1-4	Test 5 value	Test 6 value
POPULATION_SIZE	500		
TOURNAMENT_SIZE	4		
FOOD_LEFT_WEIGHT	10000000		
FOOD_UNFOUND_WEIGHT	100000		
TIME_WEIGHT	1		
NODE_WEIGHT	10000		
NODE_GROUP	50		
LEAF_ODDS	9		
NON_LEAF_ODDS	6		
MUTATE_ODDS	100		
PERCENT_GREEDY_MUTATE	80		
MAX_TURNS	1000		2000
NUM_ANTS	15		
GOAL_FITNESS	0		
POP_SEED	0 or 1		
INTERP_SEED	0		
PHEROMONE_STRENGTH	300	600	
PHEROMONE_SPREAD	50		
WAIT_ODDS	0		
AUTO_FOOD_DROP	1		
DEBUG	0		
SHOW_TIME	100		
DUMP	10000		
MAX_ITERATIONS	5000	2500	2500
RESTORE_ITERATION	0		
RESTORE_VERSION	0		
RESTORE_AMOUNT	0		
INSTRUCTION_VERSION	1,3,5,7	1	11
MAP_VERSION	1		
VERSION	1-8	9-10	11-12

12.2.2 Output Data

These are the results for tests 5 (version 09 and 10) compared with the results of test 1, for 2500 iterations.



The reason for this test was to get the ants to use pheromones by increasing the strength of the pheromones that the ants release. The best solutions in test 1 did not rely on pheromones to solve the problem. In test 5, the average fitness results are not much different than those of test 1.



Looking at the best solutions for both tests, test 5 seed 1 found a perfect solution the fastest, and test 5 seed 0 found a perfect solution the slowest of all 4 tests. This does not show that stronger pheromones worked any better.

Looking at the final best ant programs for both seeds for test 5. *Italic* statements are unnecessary.

Test 5 - Stronger Pheromones - seed 0

```
Iteration =  2491, Time = Tue Feb 18 21:12:41, Fitness =     778,
     Number of Nodes =  15, Number of Turns =  778,
     Food Left =  0, Food Not Found =  0
PICK UP FOOD
IF ( MOVE TO AWAY PHEROMONE ) THEN
  PICK UP FOOD
  PICK UP FOOD
ELSE
  IF ( CARRYING FOOD ) THEN
    MOVE TO NEST
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        PICK UP FOOD
      ELSE
        MOVE QUASI RANDOM
  PICK UP FOOD
```

This solution is basically a near perfect solution if pheromones were not being used. Unfortunately, the whole point of the test was to get the ants to use pheromones to find food faster.

Test 5 - Stronger Pheromones - seed 1

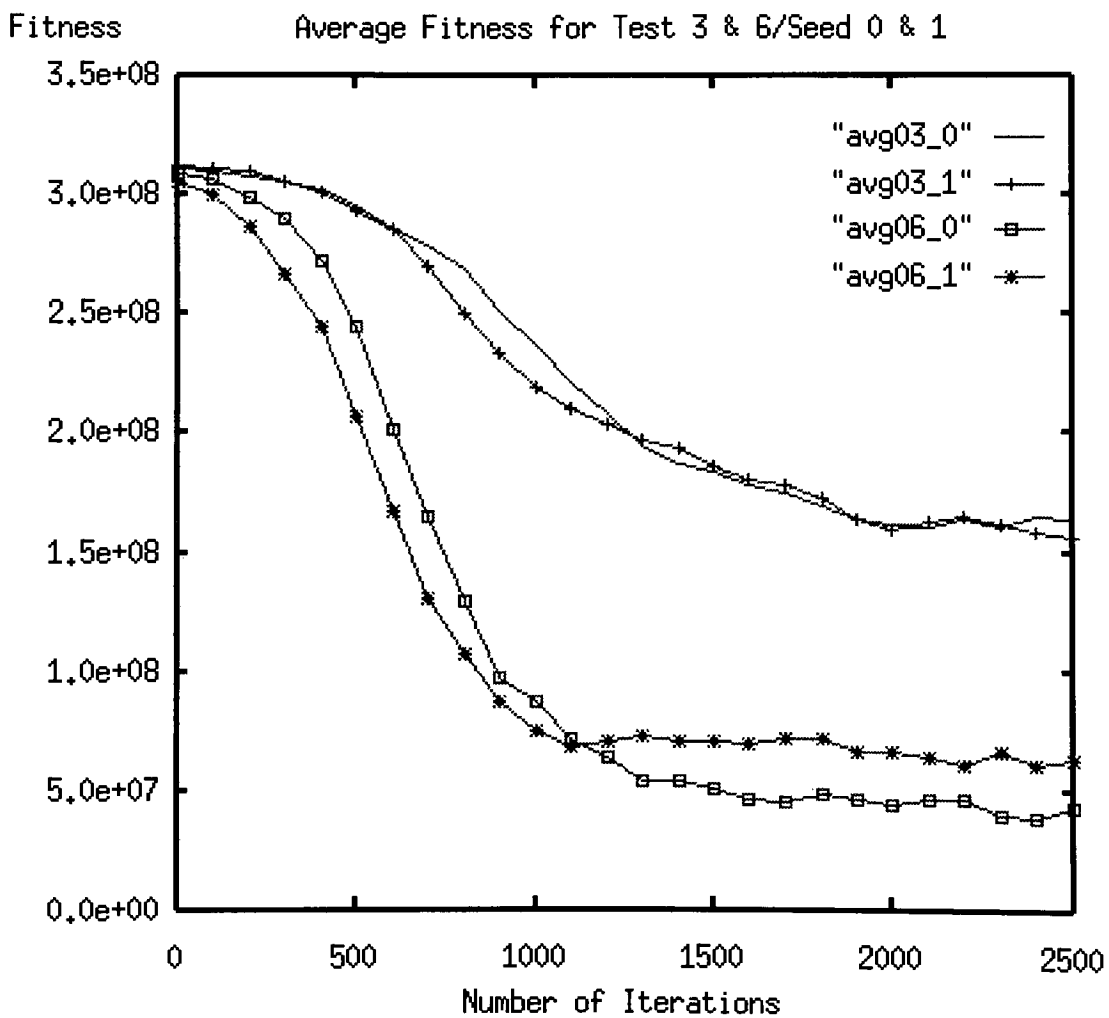
```
Iteration =  1719, Time = Tue Feb 18 21:07:34, Fitness =     746,
     Number of Nodes =  45, Number of Turns =  746,
     Food Left =  0, Food Not Found =  0
```

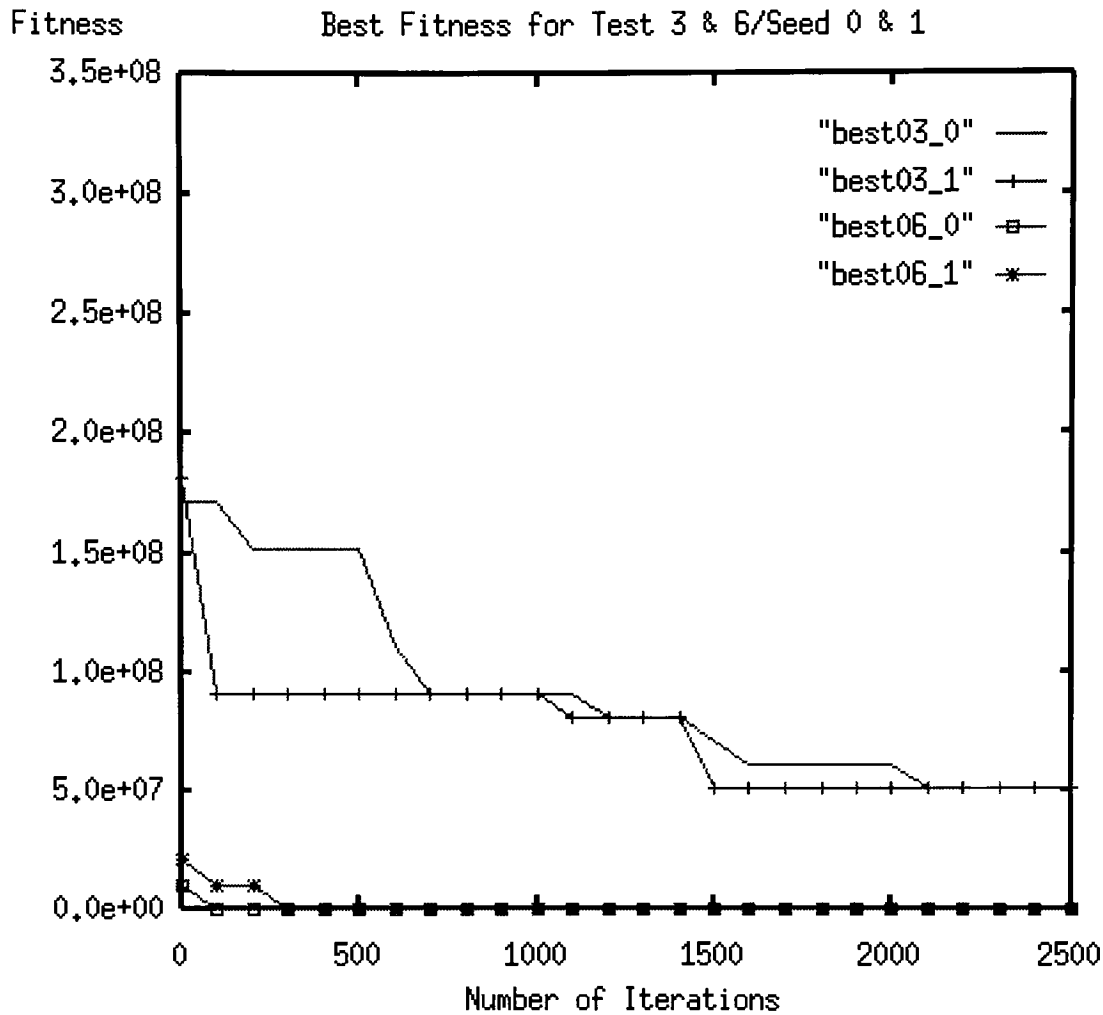
```
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      IF ( MOVE TO ADJACENT FOOD ) THEN
        PICK UP FOOD
      ELSE
        RELEASE PHEROMONE
        PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
      RELEASE PHEROMONE
      ELSE
        IF ( CARRYING FOOD ) THEN
        PICK UP FOOD
        ELSE
          IF ( MOVE TO AWAY PHEROMONE ) THEN
            MOVE QUASI RANDOM
          ELSE
            IF ( MOVE TO ADJACENT FOOD ) THEN
              PICK UP FOOD
              IF ( CARRYING FOOD ) THEN
                IF ( MOVE TO ADJACENT FOOD ) THEN
                  RELEASE PHEROMONE
                ELSE
                  IF ( MOVE TO AWAY PHEROMONE ) THEN
                    MOVE QUASI RANDOM
                  ELSE
                    MOVE QUASI RANDOM
              ELSE
                IF ( MOVE TO AWAY PHEROMONE ) THEN
                  PICK UP FOOD
                ELSE
                  MOVE TO NEST
            ELSE
              RELEASE PHEROMONE
            PICK UP FOOD
            IF ( MOVE TO ADJACENT FOOD ) THEN
              IF ( MOVE TO ADJACENT FOOD ) THEN
                MOVE QUASI RANDOM
                MOVE TO NEST
              ELSE
                IF ( CARRYING FOOD ) THEN
                  IF ( MOVE TO ADJACENT FOOD ) THEN
                    PICK UP FOOD
                  ELSE
                    RELEASE PHEROMONE
                ELSE
                  RELEASE PHEROMONE
            ELSE
              RELEASE PHEROMONE
    PICK UP FOOD
  ELSE
```

Looking at this ant program, it is hard to figure out what it does. Using the display program to test the ant program, it appears that the ants move randomly, searching for food. When the ant finds food, it releases a pheromone once, and then takes the food back to the nest. The pheromone is not big enough to attract the attention of other ants.

Neither seed for test 5 resulted in an ant program that used a pheromone trail to connect the pile of food and the nest. More changes need to be made, so that the pheromones will become useful to the ants.

The results of Test 6, which tried to get MOVE RANDOM to work as well as MOVE QUASI RANDOM by increasing *MAX_TURNS*, have been compared with the original MOVE RANDOM test (3).





Both of these plots indicate that Test 6 was much more successful that Test 3 by a wide margin. The MOVE QUASI RANDOM statement works the same as the MOVE RANDOM statement, but it allows the ants to find the food quicker. This allows all of the food to be collected is a smaller amount of time. With twice the amount of turns as test 3, test 6 was able to find all of the food.

The final best solutions of test 6 are:

Test 6 - MOVE RANDOM with longer time limit - seed 0

```
Iteration =  1726, Time = Tue Feb 18 21:04:52, Fitness =     1430,
     Number of Nodes =  25, Number of Turns = 1430,
     Food Left =  0, Food Not Found =  0
PICK UP FOOD
IF ( CARRYING FOOD ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE RANDOM
  ELSE
    MOVE TO NEST
  MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
      MOVE RANDOM
    ELSE
      PICK UP FOOD
      MOVE TO NEST
      MOVE RANDOM
  ELSE
    IF ( CARRYING FOOD ) THEN
      MOVE RANDOM
    ELSE
      IF ( MOVE TO ADJACENT FOOD ) THEN
        PICK UP FOOD
      ELSE
        PICK UP FOOD
      MOVE RANDOM
```

Test 6 - MOVE RANDOM with longer time limit - seed 1

```
Iteration =  818, Time = Tue Feb 18 20:10:36, Fitness =     1412,
     Number of Nodes =  15, Number of Turns = 1412,
     Food Left =  0, Food Not Found =  0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
  IF ( CARRYING FOOD ) THEN
    MOVE TO NEST
    MOVE TO NEST
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      MOVE RANDOM
    ELSE
      PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( CARRYING FOOD ) THEN
      MOVE RANDOM
    ELSE
      PICK UP FOOD
  ELSE
    MOVE RANDOM
```

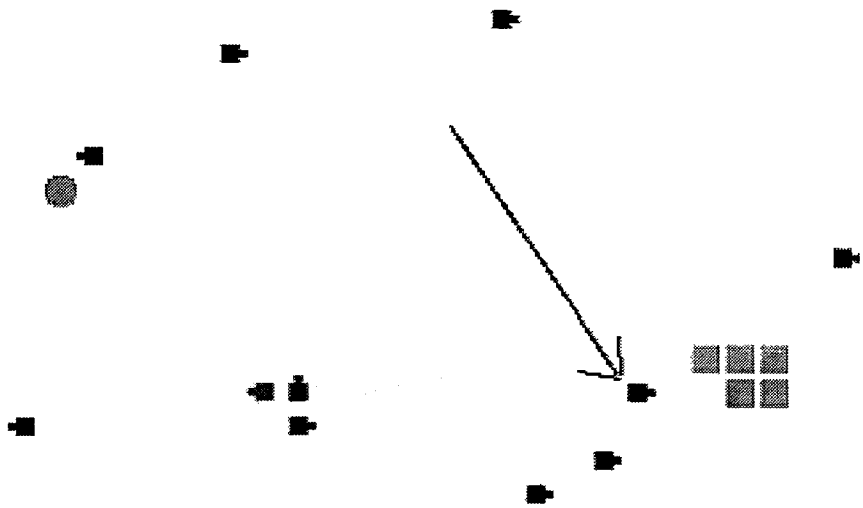
12.2.3 Synopsis

Test 6 was a success. It proved the theory of why MOVE QUASI RANDOM works much better than MOVE RANDOM. The MOVE QUASI RANDOM statement can be safely used in later tests, as a means of shortening the time required by the ants to solve a problem.

Test 5 failed just as test 1 failed in getting the ants to use pheromones. To find the reason for this I wrote what I believed to be the best solution, and tested it with the display program. This is the program I tested:

```
IF ( CARRYING FOOD ) THEN
    RELEASE PHEROMONE
    MOVE TO NEST
ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
        PICK UP FOOD
    ELSE
        IF ( MOVE TO AWAY PHEROMONE ) THEN
            PICK UP FOOD
        ELSE
            MOVE QUASI RANDOM
```

When I tested this program, I found a bug with the IF (MOVE TO AWAY PHEROMONE) THEN statement. What I found was that when an ant is to the South-East of the nest, and is following pheromones, then the two directions away from the nest that it can go are South and East. It will go in the direction of the strongest pheromone. But, when the South and East square both have pheromones of equal strength, then the ant will always go South. In the example below, this means that the marked ant will not find the pile of food.



In the previous example, one ant has found the lower left piece of food in the top right pile of food. It releases a pheromone in the square where it found the food, and moves toward the nest, releasing pheromones in each square. Other ants have picked up on the trail, but the trail has already started to dissipate. When a pheromone spreads, the square below the pheromone, and to the right of the pheromone will both have the same strength. Even though the food is to the right, the ant will go South, because when pheromones are equal strength, the code is written so that the ant will move South. Thus, the marked ant will move one space forward, but will then turn South, missing the food.

To fix this problem, I have rewritten the IF (MOVE TO AWAY PHEROMONE) THEN statement, so that in case of equal pheromone strengths, the ant will continue moving in the current direction that it is facing. Thus, in this case the ant will head East, not South. If the ant is not facing in either of the directions away from the nest, then it will pick the direction that is only one turn away.

In testing this ant program with the display program, I have also found that the best values for pheromones are:

PHEROMONE_STRENGTH = 300
PHEROMONE_SPREAD = 20

The *PHEROMONE_STRENGTH* did not need to be increased as it was in test 5.

12.3 Simple Food Collection Problem (Test 7)

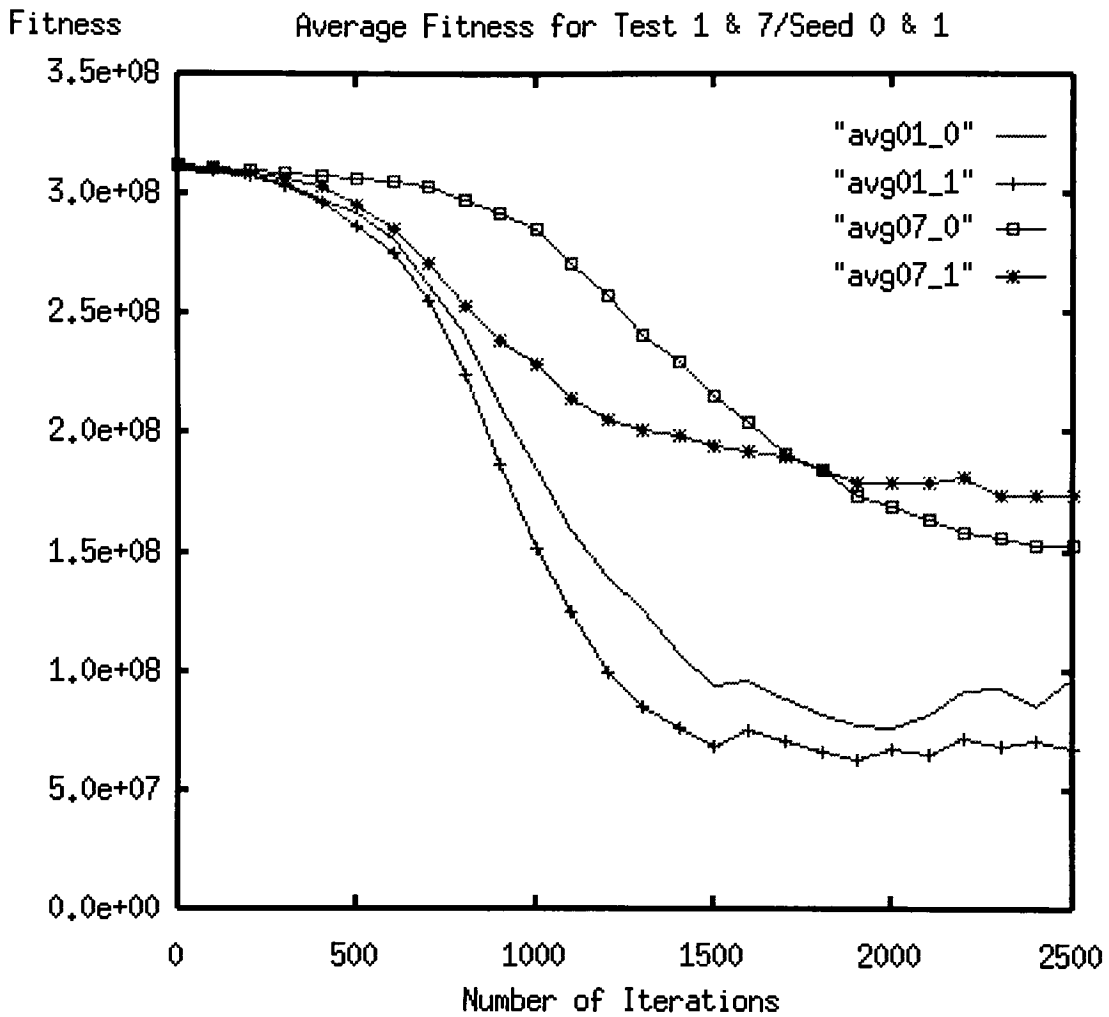
12.3.1 Input Parameters

One more test for the Simple Food Collection Problem was run in an attempt to get the ants to use pheromones. This test (7) is the similar to tests 1 and 5. The pheromone values have been changed, using what was learned from the test with the display program. The *NUM_TURNS* value has also been decreased, so that it will be harder for the ants to solve the problem, without the use of pheromones.

Parameter	Tests 1	Test 5 value	Test 7 value
POPULATION_SIZE	500		
TOURNAMENT_SIZE	4		
FOOD_LEFT_WEIGHT	10000000		
FOOD_UNFOUND_WEIGHT	100000		
TIME_WEIGHT	1		
NODE_WEIGHT	10000		
NODE_GROUP	50		
LEAF_ODDS	9		
NON_LEAF_ODDS	6		
MUTATE_ODDS	100		
PERCENT_GREEDY_MUTATE	80		
MAX_TURNS	1000		500
NUM_ANTS	15		
GOAL_FITNESS	0		
POP_SEED	0 or 1		
INTERP_SEED	0		
PHEROMONE_STRENGTH	300	600	300
PHEROMONE_SPREAD	50		20
WAIT_ODDS	0		
AUTO_FOOD_DROP	1		
DEBUG	0		
SHOW_TIME	100		
DUMP	10000		
MAX_ITERATIONS	5000	2500	2500
RESTORE_ITERATION	0		
RESTORE_VERSION	0		
RESTORE_AMOUNT	0		
INSTRUCTION_VERSION	1	1	1
MAP_VERSION	1		
VERSION	1-2	9-10	13-14

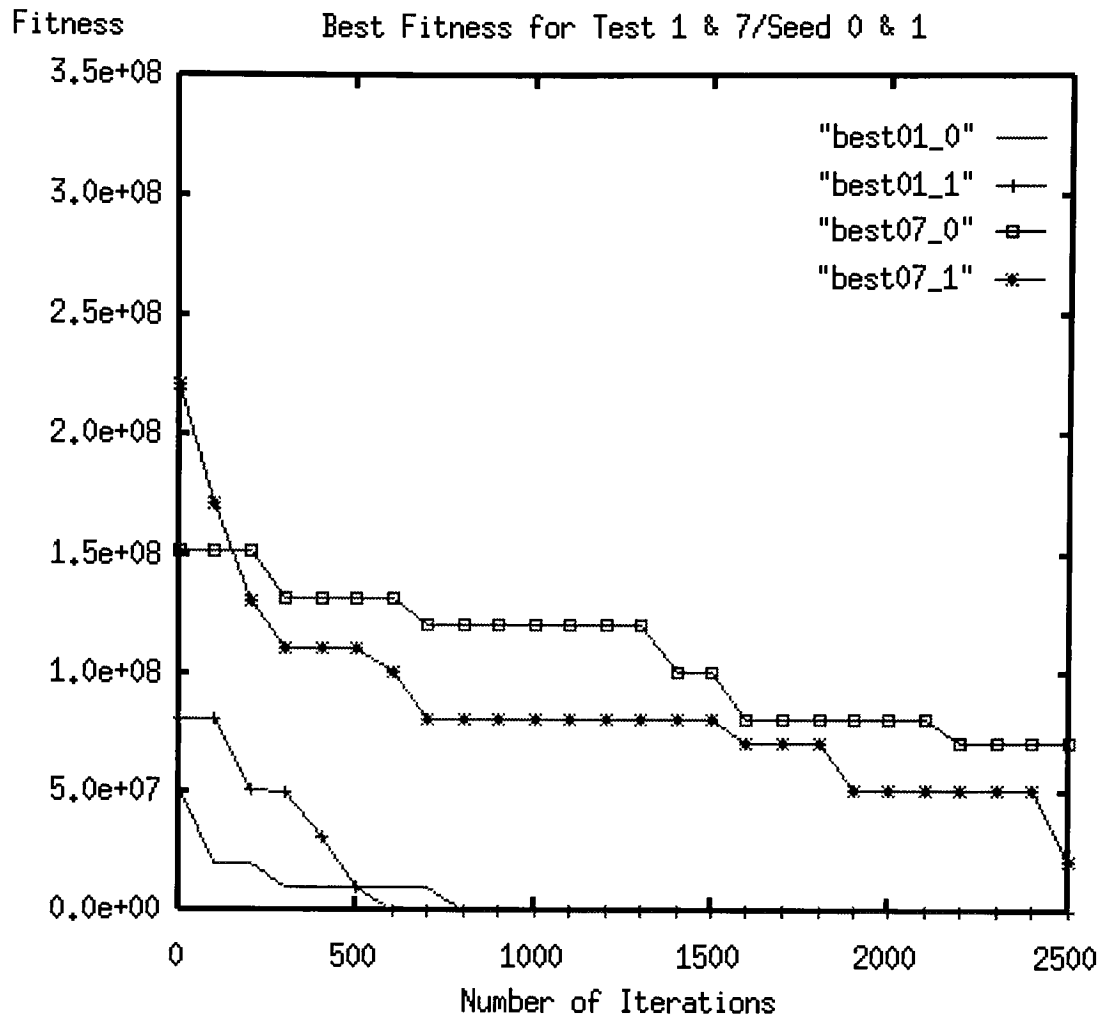
12.3.2 Output Data

This test (7) did much better that the previous tests. Although it did not find a perfect solution that found all of the food, if given more turns, the ants could have easily found all of the food.



Because the MAX_TURNS was set so low, test 7 appears to have done worse that the original test 1. Between the two runs on test 7, the average ant program left about 17 pieces of food. The ants in this program had 500 turns to find the food. For test 1, the average food left was around 8, and the ants were given 1000 turns to find the food.

If a standard progression was assumed for test 7, then it appears that test 7 was slightly worse than test 1. When the time for the ants to spread out and initially find the first piece of food (at least 50 turns) is factored in, then test 7 provides better results than test 1.



Again, test 7 appears worse than test1. Still, the best solution from test 7 seed 1 found all but 2 pieces in 500 turns. Looking back to the results from test 1, the best ant solutions for both seeds found all of the food in 734 and 692 turns. And, test 1 was run for 5000 iterations, twice as many as test 7. That means that the programs were evaluated more often will different seeds, giving it more opportunities to find the food faster.

The best ant programs produced by test 7 are as follows:

Test 7 Third attempt to get ants to use pheromones - seed 0

Iteration = 2113, Time = Wed Mar 5 19:10:36, Fitness = 70700500,
Number of Nodes = 41, Number of Turns = 500,
Food Left = 7, Food Not Found = 7

```
IF ( MOVE TO ADJACENT FOOD ) THEN
  IF ( CARRYING FOOD ) THEN
    MOVE QUASI RANDOM
  ELSE
    IF ( CARRYING FOOD ) THEN
      MOVE TO NEST
    ELSE
      PICK UP FOOD
ELSE
  IF ( CARRYING FOOD ) THEN
    IF ( MOVE TO ADJACENT FOOD ) THEN
      IF ( CARRYING FOOD ) THEN
        MOVE QUASI RANDOM
      ELSE
        IF ( CARRYING FOOD ) THEN
          RELEASE PHEROMONE
        ELSE
          MOVE QUASI RANDOM
    ELSE
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        MOVE QUASI RANDOM
      ELSE
        MOVE TO NEST
  ELSE
    PICK UP FOOD
    IF ( CARRYING FOOD ) THEN
      MOVE QUASI RANDOM
    ELSE
      MOVE QUASI RANDOM
IF ( CARRYING FOOD ) THEN
  IF ( CARRYING FOOD ) THEN
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      IF ( CARRYING FOOD ) THEN
        MOVE QUASI RANDOM
      ELSE
        MOVE QUASI RANDOM
    ELSE
      MOVE TO NEST
  ELSE
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        IF ( CARRYING FOOD ) THEN
          IF ( CARRYING FOOD ) THEN
            MOVE TO NEST
          ELSE
            PICK UP FOOD
        ELSE
          MOVE QUASI RANDOM
      ELSE
        IF ( MOVE TO AWAY PHEROMONE ) THEN
          MOVE TO NEST
        ELSE
```

```
        RELEASE PHEROMONE
ELSE
  PICK UP FOOD
```

Test 7 - Third attempt to get ants to use pheromones - seed 1

```
Iteration =  2481, Time = Wed Mar  5 19:20:33, Fitness =  20200500,
     Number of Nodes =  9, Number of Turns =  500,
     Food Left =  2, Food Not Found =  2
IF ( CARRYING FOOD ) THEN
  RELEASE PHEROMONE
  MOVE TO NEST
ELSE
  IF ( MOVE TO AWAY PHEROMONE ) THEN
    PICK UP FOOD
  ELSE
    MOVE QUASI RANDOM
  PICK UP FOOD
```

The first program has many unnecessary statements. There are no RELEASE PHEROMONE statements that can be executed by the ant, so this solution does not use pheromones. This was proved using the display program.

The second program is almost perfect. It contains no unnecessary statements. It leaves only 2 pieces of food in 500 turns. The problem is that it does not use the IF (MOVE TO ADJACENT FOOD) THEN statement. This means that it must randomly find food by moving to the same square as the food.

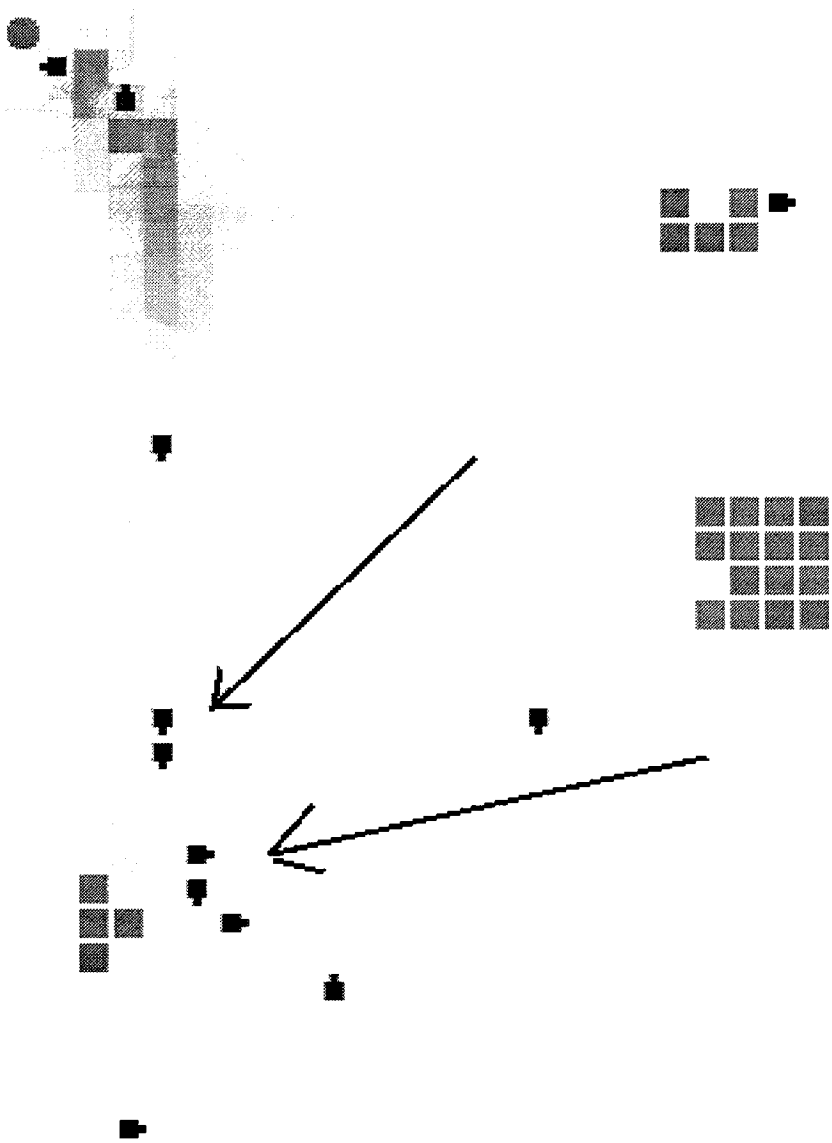
12.3.3 Synopsis

Testing this solution with the display program many times shows that almost never finds the food within 1000 turns. The reason is that the pheromones trails build up so wide, that they start leading the ants away from the food. Trails will still exist for food piles that have already been completely depleted.

Another problem is that when the right side of a pile of food is found first, the pheromone trail could lead the ants past the left side of the pile of food. If the ants cannot detect food next to them, as in the test 7 results, then they will not be able to find the food. This can be seen happening in the following picture example.

Here, the ants are all following the trail to the bottom left pile of food. There were trails to the other piles of food, but they have dissipated. The 4 remaining pieces of the bottom left pile of food are the left most pieces. The first set of 2 ants marked by the arrow are heading 1 square to the right of the food. By following the pheromone trail, they will end up where the 3 ants marked by the second arrow are. The ants are following pheromones, and moving away from the nest. Thus, they cannot move left until they are outside the pher-

omone trail, because the nest is to the left of the ants. Because of the order that the other pieces of food in the pile were found, the squares to the right have a stronger pheromone scent than those to the bottom. Therefore, the ants will be turned right, away from the food.



Eventually, the pheromone trail will dissipate, and other ants will be able to find the food. When this happens, many ants will be close to the food, because they followed the pheromone trail. At least one ant will quickly find the food again.

12.4 Water Crossing Problem (Test 8-9)

12.4.1 Input Parameters

In this test, there is a stream of water 2 blocks wide completely separating the ants and the food. The ants must move in to the water and die to build a bridge of dead ants, so that other ants can cross the water to get the food.

This Water Crossing Test was run 4 times, two tests with two different seeds. The official test numbers for testing even between different problems will not duplicate. The test numbers for these two tests are 8 and 9. The parameters were:

	Test 8	Test 9
POPULATION_SIZE	500	
TOURNAMENT_SIZE	4	
FOOD_LEFT_WEIGHT	10000000	
FOOD_UNFOUND_WEIGHT	100000	
TIME_WEIGHT	1	
NODE_WEIGHT	10000	
NODE_GROUP	50	
LEAF_ODDS	9	
NON_LEAF_ODDS	6	
MUTATE_ODDS	100	
PERCENT_GREEDY_MUTATE	80	
MAX_TURNS	500	1000
NUM_ANTS	50	100
GOAL_FITNESS	0	
POP_SEED	0 & 1	
INTERP_SEED	0	
PHEROMONE_STRENGTH	400	
PHEROMONE_SPREAD	20	
WAIT_ODDS	0	
AUTO_FOOD_DROP	1	
DEBUG	0	
SHOW_TIME	100	
DUMP	10000	
MAX_ITERATIONS	2500	
RESTORE_ITERATION	0	
RESTORE_VERSION	0	
RESTORE_AMOUNT	0	
INSTRUCTION_VERSION	15	
MAP_VERSION	15	
VERSION	15-16	17-18

The tests used basically the same parameters from the last test for the Simple Food Collection Problem. The number of ants was increased to 50 and 100, because many ants will die. Also, in the ninth test, the number of turns to complete the problem was increased to 1000.

The instruction set for this problem had to be increased to handle statements for crossing water. The instruction set for both tests was the same. It is probably larger than need be, with 12 different instructions. The purpose of this test is to determine what instructions the ants might find useful. The instruction set was:

```
MOVE QUASI RANDOM
MOVE TO NEST THROUGH WATER
PICK UP FOOD
RELEASE PHEROMONE
MOVE INTO WATER

PROC2
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
IF ( MOVE TO DEAD ANT ) THEN
IF ( WATER AHEAD ) THEN
IF ( ANOTHER ANT HERE ) THEN
```

The MOVE TO NEST THROUGH WATER statement can be useful for ants carrying food back to the nest. The first couple of ants with food will build a bridge, connecting the food and nest at the closest point. These ants will drop their food at the ant bridge where they die, so other ants will find the food at the bridge.

The MOVE INTO WATER statement is necessary for the ants to first cross the water to find the food.

The IF (MOVE TO DEAD ANT) THEN, IF (WATER AHEAD) THEN. and IF (ANOTHER ANT HERE) THEN statements can each be used to determine when an ant should move into water. The schema

```
IF ( MOVE TO DEAD ANT ) THEN
        MOVE INTO WATER
```

will cause an ant to move into the water when it moves to a dead ant. Since the water stream is only 2 blocks wide, and the MOVE INTO WATER statement only moves the ant forward, then this schema will complete any bridge that has already been started. But, the ants must have a way to start the bridge, without having every ant kill itself. The schema:

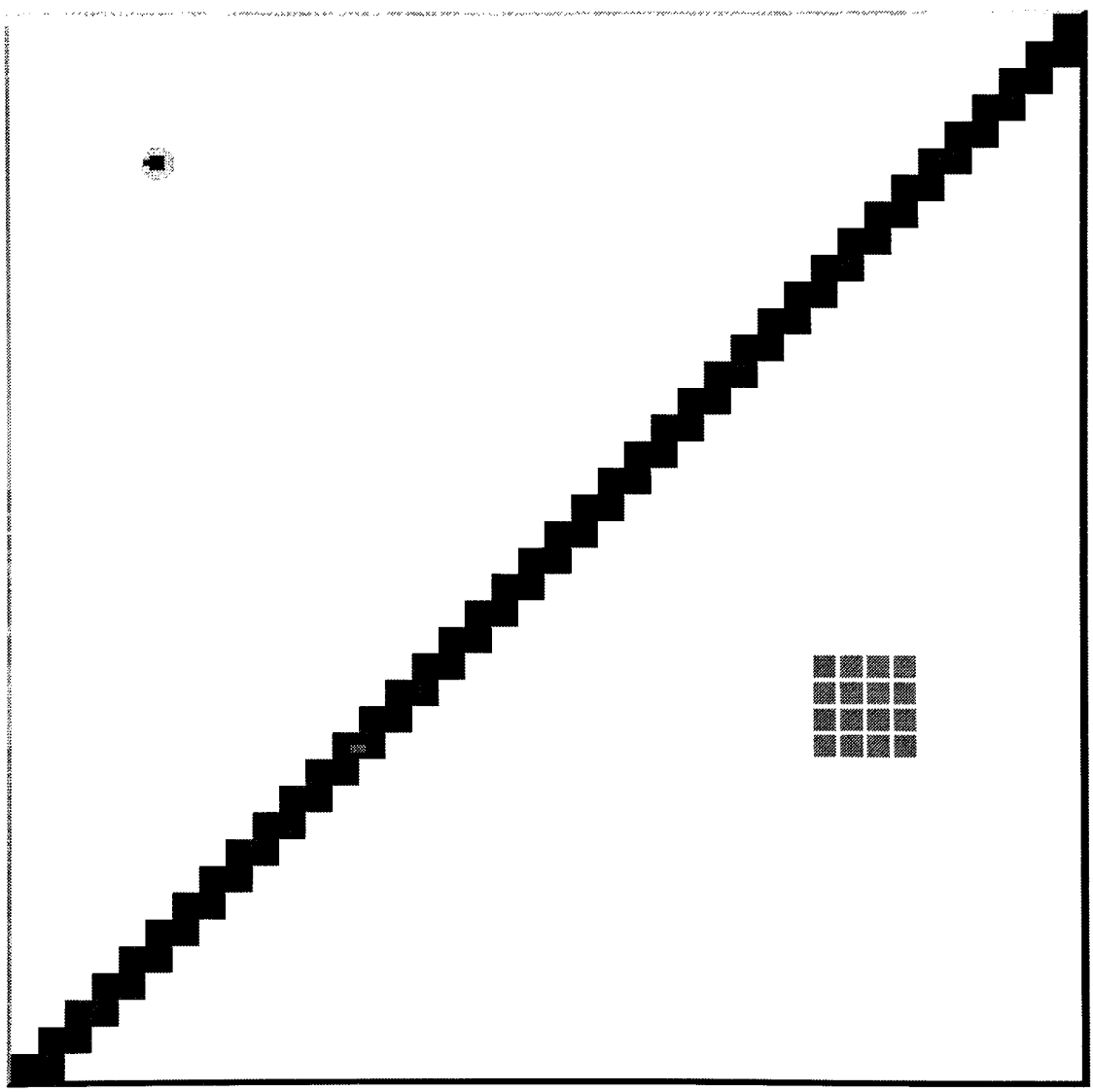
```
IF ( ANOTHER ANT HERE ) THEN
        MOVE INTO WATER
```

will only allow an ant to kill itself if there is another ant at the same location. By pure statistics, this will cause only a couple of ants to die, because of the odds that two ants will be together, and one will be facing the water. To increase this chance, the ants could do:

```
IF ( WATER AHEAD ) THEN
        IF ( ANOTHER ANT HERE ) THEN
                MOVE INTO WATER
        ELSE
                RELEASE PHEROMONE
ELSE
        IF ( MOVE TO AWAY PHEROMONE ) THEN
                PICK UP FOOD
        ELSE
                MOVE RANDOM
```

Here, the ants first try to move into water, then try to find other ants near water, then try to find water.

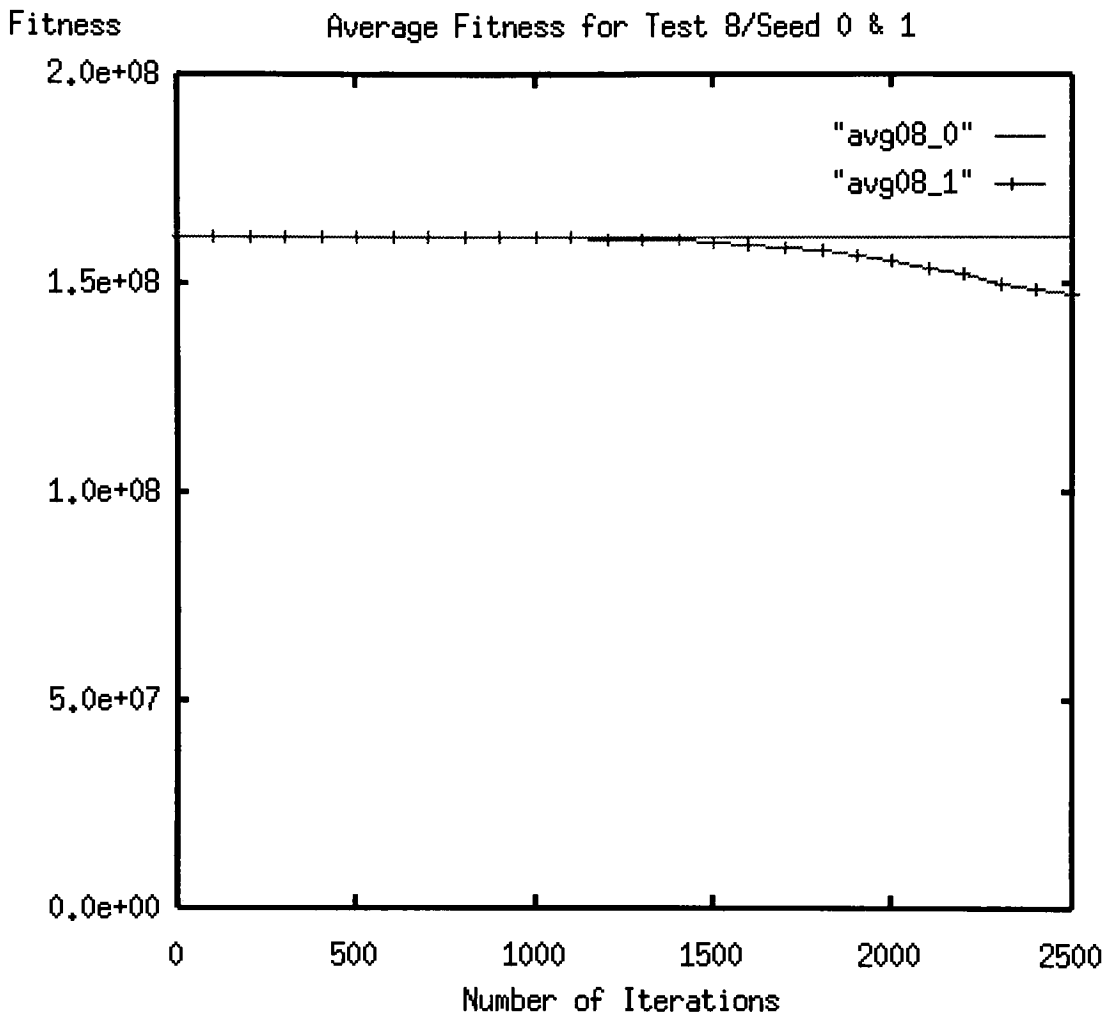
There are 16 pieces of food in the map. The map file for these tests is:

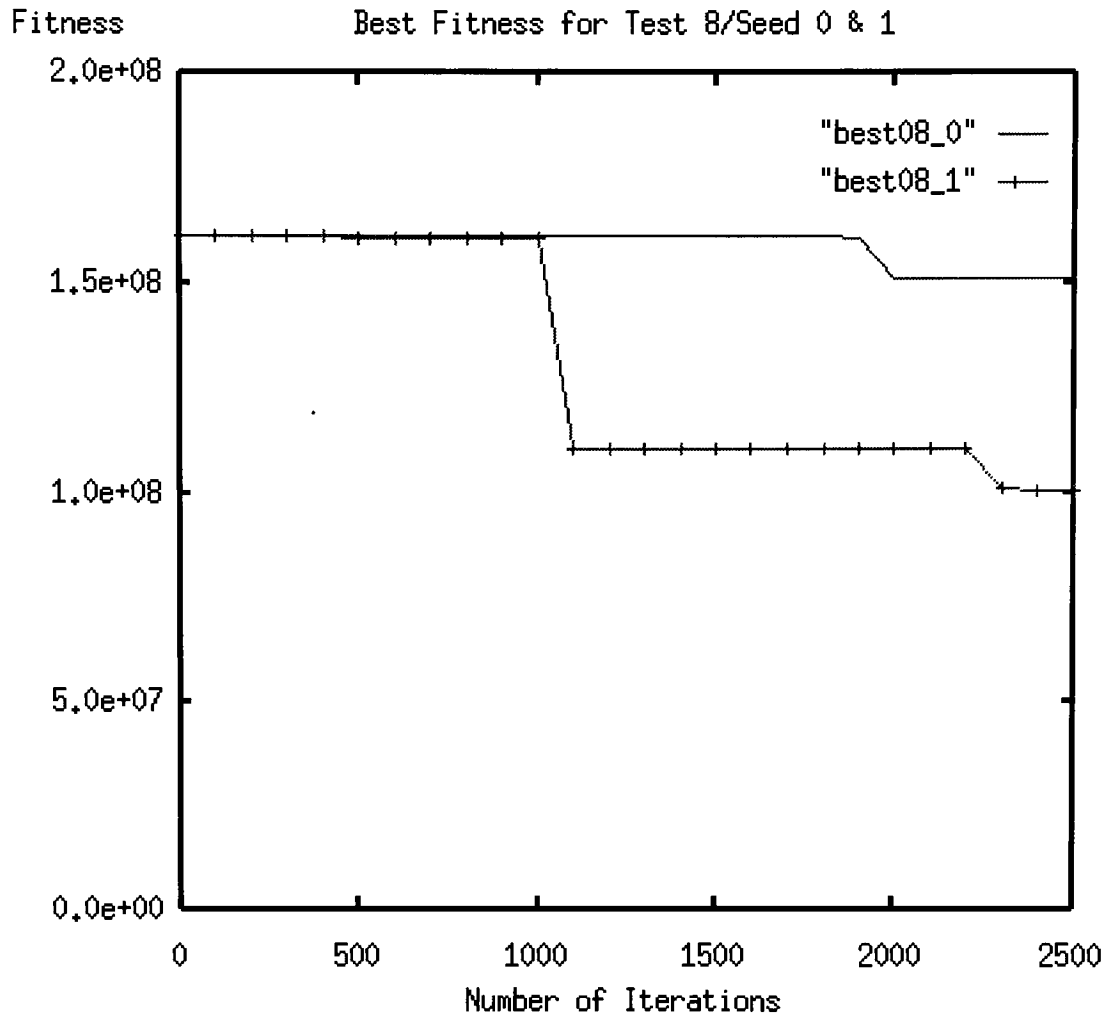


12.4.2 Output Data

Since test 8 and 9 were run for a different number of iterations, the graphs for the results of each test will be displayed separately.

The average and best fitness graphs for test 8 (50 ants 500 turns) are:





These graphs show that the test was basically a complete failure.

The programs produced by test 8 were:

Test 8- Water test with 50 ants - seed 0

```
Iteration =  1946, Time = Mon Mar 10 21:57:48, Fitness = 151100500,
     Number of Nodes =  7, Number of Turns =  500,
     Food Left = 15, Food Not Found = 11
IF ( WATER AHEAD ) THEN
  MOVE INTO WATER
ELSE
  MOVE QUASI RANDOM
MOVE QUASI RANDOM
PICK UP FOOD
```

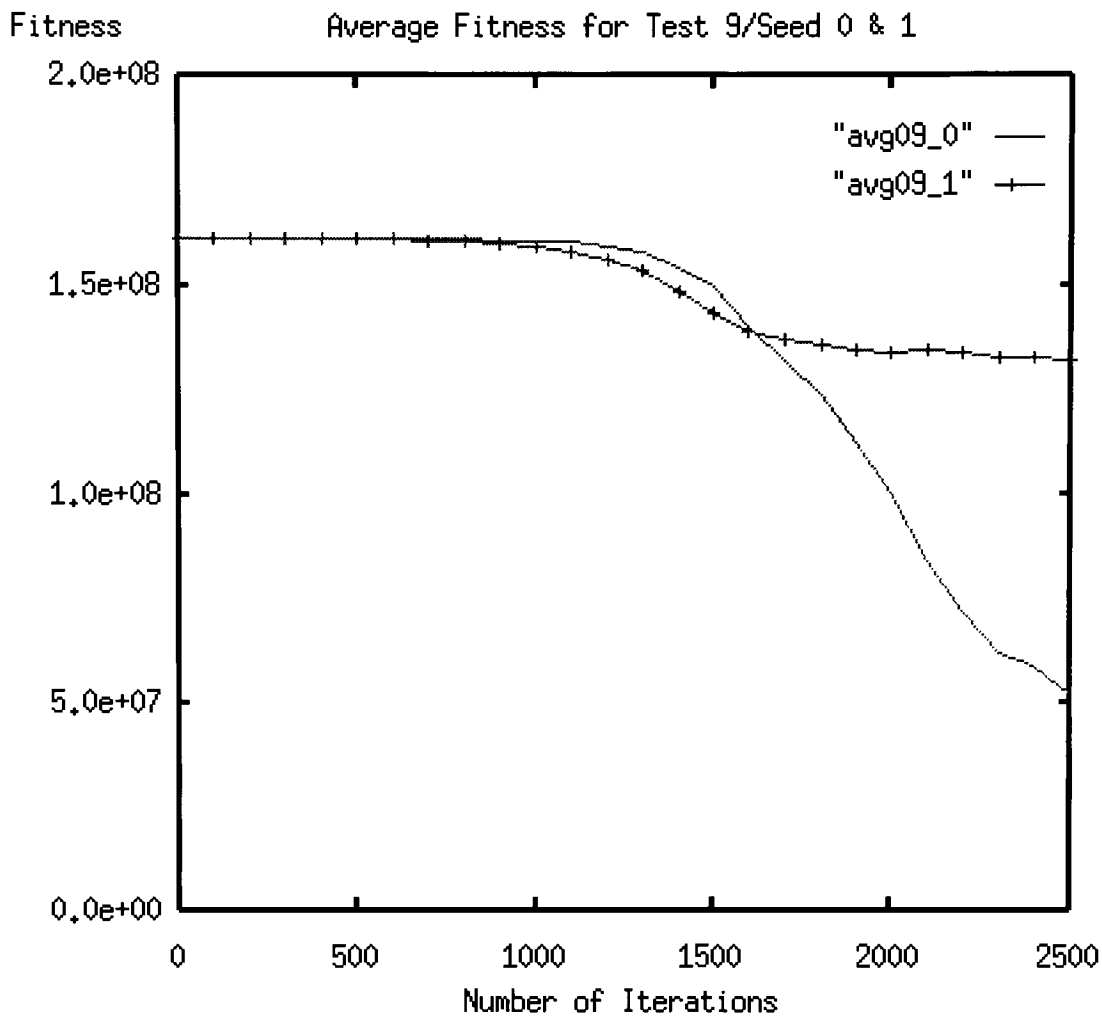
Test 8- Water test with 50 ants - seed 1

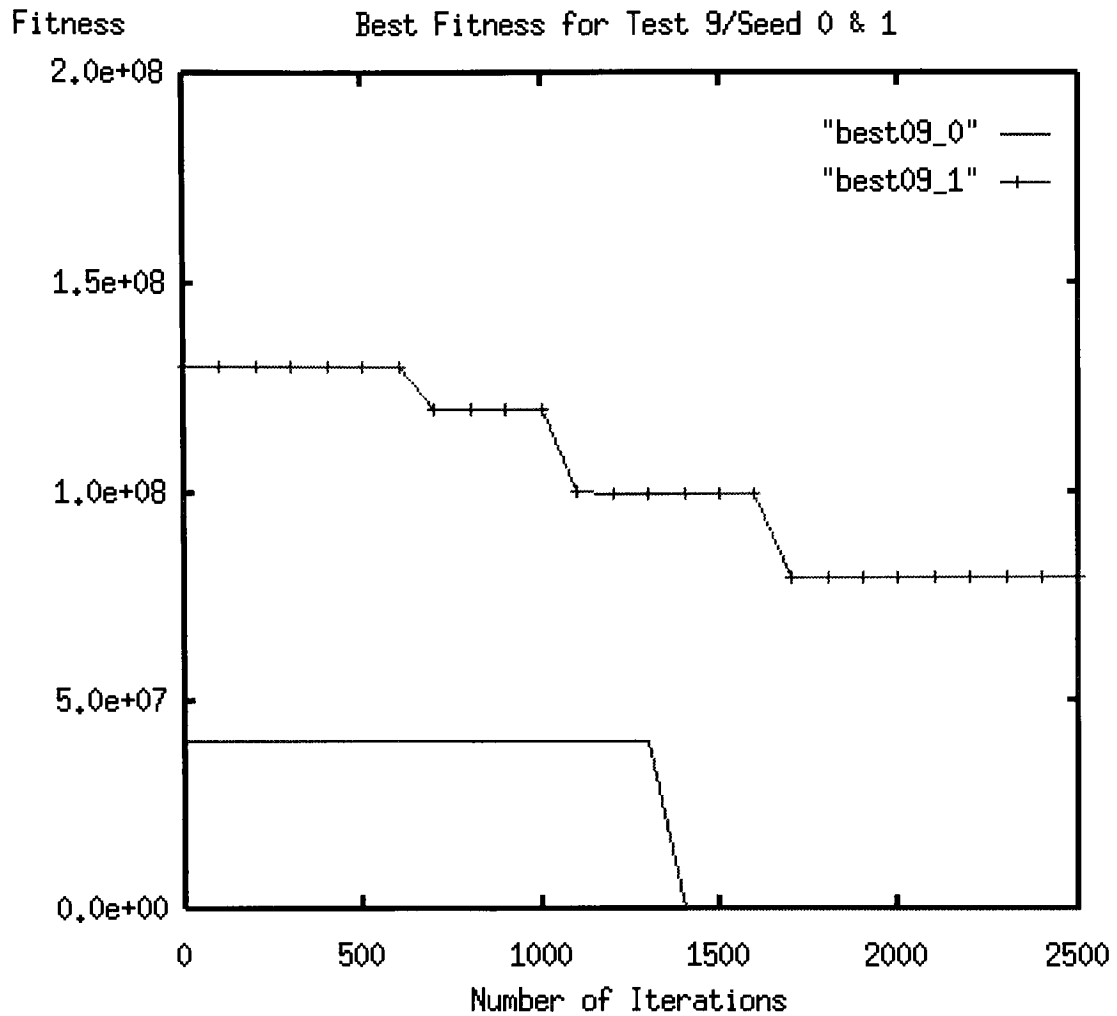
```
Iteration =  2400, Time = Mon Mar 10 22:25:23, Fitness = 100900500,
     Number of Nodes =  17, Number of Turns =  500,
     Food Left = 10, Food Not Found =  9
IF ( MOVE TO AWAY PHEROMONE ) THEN
  IF ( MOVE TO DEAD ANT ) THEN
    MOVE INTO WATER
  ELSE
    PICK UP FOOD
ELSE
  IF ( CARRYING FOOD ) THEN
    MOVE TO NEST THROUGH WATER
  ELSE
    MOVE INTO WATER
  IF ( CARRYING FOOD ) THEN
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      MOVE QUASI RANDOM
    ELSE
      MOVE TO NEST THROUGH WATER
  ELSE
    MOVE QUASI RANDOM
    IF ( MOVE TO ADJACENT FOOD ) THEN
      RELEASE PHEROMONE
    ELSE
      MOVE INTO WATER
```

For test 8 seed 0, basically the genetic program got lucky once, and one ant randomly managed to find food and randomly bring it back to the nest without killing itself. Using the display program to test this solution, it can be seen that almost all of the water that the ants move into is in the middle of the map. By the time an ant finds a piece of food, there are only 4 ants left alive.

Test 8 seed 1 performed slightly better. The ants actually try to move to the nest when they found food, although they still move into water whenever they can find it. The only place the ants release pheromones is when they are at the pile of food. Then the ant walks directly to the nest. In test 8 seed 0, in the best result, 1 piece of food was taken to the nest, and another 4 had been found, but not taken to the nest. The improvement for this test is simply that it brings the food back to the nest faster.

The results for test 9 were:





The programs produced by this test are:

Test 9 - Water Problem with 100 ants - seed 0

```
Iteration =  2154, Time = Thu Mar 13 00:10:31, Fitness =    10297,
     Number of Nodes =  53, Number of Turns =  297,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO DEAD ANT ) THEN
  RELEASE PHEROMONE
ELSE
  PICK UP FOOD
IF ( MOVE TO AWAY PHEROMONE ) THEN
  IF ( CARRYING FOOD ) THEN
    IF ( WATER AHEAD ) THEN
      MOVE QUASI RANDOM
    ELSE
      MOVE TO NEST THROUGH WATER
      IF ( MOVE TO ADJACENT FOOD ) THEN
        MOVE TO NEST THROUGH WATER
```

```
      ELSE
        RELEASE PHEROMONE
  ELSE
    IF ( CARRYING FOOD ) THEN
    MOVE QUASI RANDOM
    ELSE
      IF ( MOVE TO ADJACENT FOOD ) THEN
        IF ( WATER AHEAD ) THEN
          IF ( MOVE TO ADJACENT FOOD ) THEN
            PICK UP FOOD
          ELSE
            PICK UP FOOD
        ELSE
          RELEASE PHEROMONE
      ELSE
        IF ( CARRYING FOOD ) THEN
          IF ( ANOTHER ANT HERE ) THEN
            MOVE INTO WATER
          ELSE
            MOVE QUASI RANDOM
        ELSE
          IF ( WATER AHEAD ) THEN
            MOVE TO NEST THROUGH WATER
          ELSE
            RELEASE PHEROMONE
      IF ( CARRYING FOOD ) THEN
        IF ( MOVE TO DEAD ANT ) THEN
          IF ( WATER AHEAD ) THEN
            MOVE TO NEST THROUGH WATER
          ELSE
            MOVE QUASI RANDOM
        ELSE
          IF ( MOVE TO ADJACENT FOOD ) THEN
            IF ( MOVE TO AWAY PHEROMONE ) THEN
              IF ( CARRYING FOOD ) THEN
                MOVE TO NEST THROUGH WATER
              ELSE
                IF ( ANOTHER ANT HERE ) THEN
                  PICK UP FOOD
                ELSE
                  MOVE TO NEST THROUGH WATER
                IF ( ANOTHER ANT HERE ) THEN
                  MOVE QUASI RANDOM
                ELSE
                  MOVE INTO WATER
            ELSE
              MOVE TO NEST THROUGH WATER
          ELSE
            MOVE INTO WATER
      ELSE
        PICK UP FOOD
ELSE
  MOVE INTO WATER
IF ( MOVE TO ADJACENT FOOD ) THEN
```

```
  RELEASE PHEROMONE
ELSE
  MOVE QUASI RANDOM
```

Test 9 - Water Problem with 100 ants - seed 1

```
Iteration =  1613, Time = Wed Mar 12 22:48:40, Fitness = 80001000,
     Number of Nodes =  13, Number of Turns = 1000,
     Food Left =  8, Food Not Found =  0
IF ( MOVE TO DEAD ANT ) THEN
  MOVE INTO WATER
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO AWAY PHEROMONE ) THEN
      RELEASE PHEROMONE
    ELSE
      PICK UP FOOD
  ELSE
    IF ( WATER AHEAD ) THEN
      IF ( MOVE TO ADJACENT FOOD ) THEN
        MOVE INTO WATER
      ELSE
        IF ( MOVE TO DEAD ANT ) THEN
          MOVE INTO WATER
        ELSE
          MOVE INTO WATER
    ELSE
      MOVE QUASI RANDOM
```

Test 9 seed 0 performed very well. Using the display program, I found that the genetic program found an ingenious method of solving the problem. This is a desctipion of what was seen with the display program.

1) First, the ants moved around randomly, moving into water if it was in front of them.

2) When an ant found another dead ant, it released pheromones, and moved forward into water. This would create a bridge across a stream of width 2. This happened before 4 ants had died.

3) Other ants picked up on the pheromone scent, and crossed the water without dying. When the ants smelles pheromones, they would follow them, and never move into the water while they smelled the pheromones. The other ants also leave pheromones when they see the dead ants and smell pheromones. By the time this step had happened, only 1 more ant had died elsewhere.

4) Within a couple of turns, 10 ants have crossed the water at the one bridge, and 1 of them has found the food. That ant has picked up a piece of food, and is moving back and forth between 2 other pieces of food, releasing pheromones every turn.

5) Other ants follow these pheromones, and pick up the food under the ant moving back and forth. When this food is gone, that ant is freed to go back to the nest. It does a drunken walk to the nest.

6) On the way to the nest, if it encounters water, it will move into the water, dropping the food where it died.

In the best run from the genetic program, this solution solved the problem and found all of the food in 297 iterations.

Test 9 seed 1 did not perform well. It did not use a good method of crossing the water, and it returned food to the nest using random movements.

12.4.3 Synopsis

The two tests here produced two different solutions. One was the move into the water and die solution. Here, most of the ants kill themselves, until there is no water in the way of the surviving ants. Then the remaining ants perform the Simple Food Collection Problem. This was the solution found with both seeds in test 8 and seed 1 in test 9.

The other solution worked much better. The average and best fitness were both well below any of the other tests. Also, a minimal number of ants die using this solution. This solution has 53 nodes, which is more than *NODE_GROUP*. If the program had been 4 nodes less, then it wouldn't have been penalized for size. Its fitness would be 297, because it took 297 turns to find all of the food, instead of 10297.

If this solution had been given more time to run, it may have come up with a similar, but simpler, solution. This will be the goal of the next test.

12.5 Water Crossing Problem (Test 10)

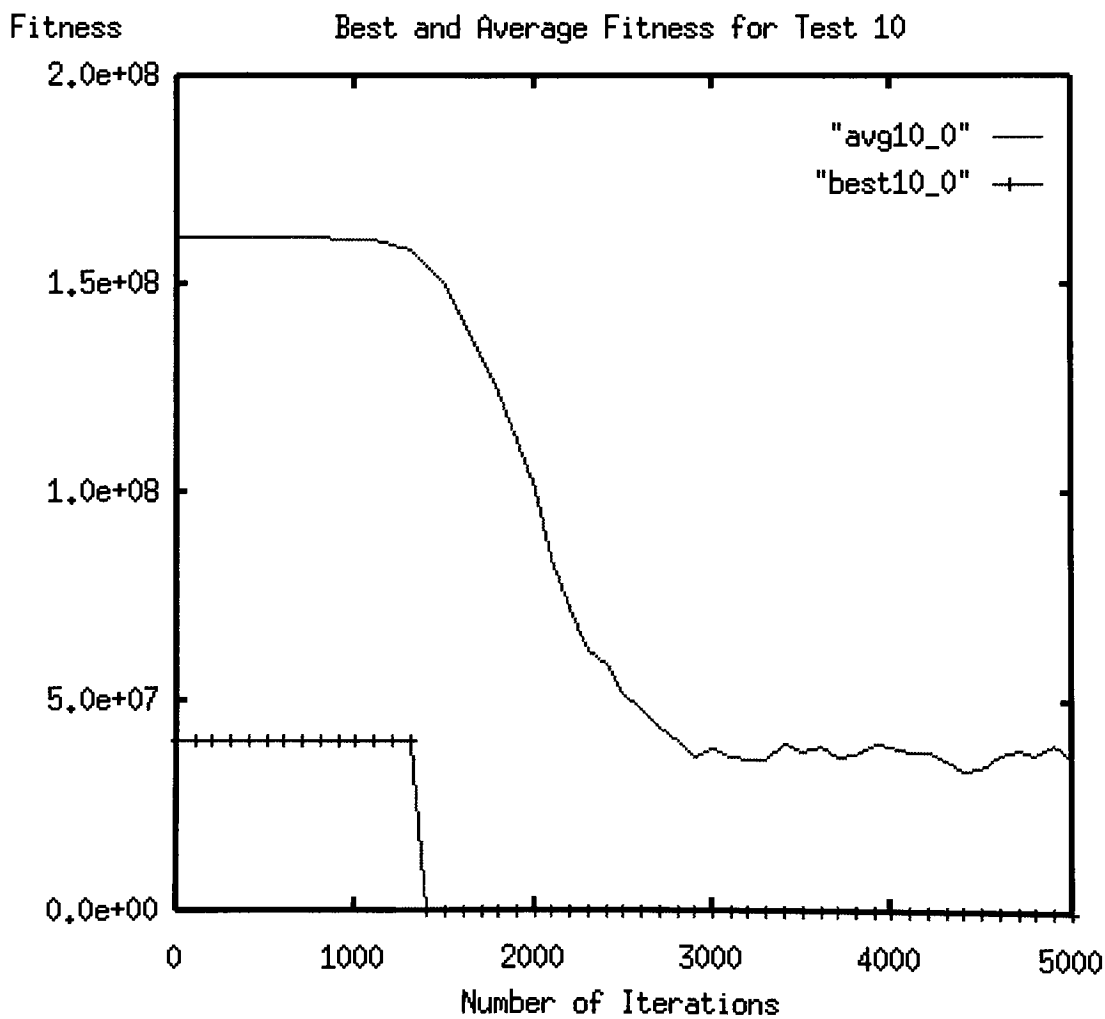
12.5.1 Input Parameters

Test 10 was run with only 1 seed. It is exactly like test 9 seed 0, except that *MAX_ITERATIONS* was extended to 5000, double the value in test 9. The first 2500 iterations will be exactly like test 9. Hopefully, after iteration 2500, the genetic program will develop a more concice solution.

12.5.2 Output Data

Test 10 found a better solution 5 times after iteration 2500. These were found at iteration 2714, 2816, 3033, 3937, and 4751. All of these solution were different, but they were all exactly 53 nodes. They found all of the food in 267, 263, 228, 205, and 162 turns respectively.

The fitness plots for best and average are:



It appears that the genetic program had done the best it could do after 2500 iterations. Most of the improvement in the best solution came from testing the same program over and over, and getting luckier with having the ants finding the food faster.

This is not true of the final solution. The best solution that it came up with is very different than the solution from test 9. It used pheromones to force the ants to spread apart.

From the very first turn, every ant releases a pheromone. The ants execute MOVE TO AWAY PHEROMONE, so they all move away from the nest as fast as possible. Basically, this means that the ants get caught in the corners. The ants that get caught in the top left and right, and bottom left corners are basically stuck, because they are still moving from the pheromones. Some of the ants are pushed toward the bottom right, directly at the water and the food. But, the ants can't move into water if there are pheromones present. This condition is still left over from the best solution in iteration 2154. Thus, for the ants to get through the water, two ants must reach the water before the pheromones that they dropped behind them can catch up with them. If this doesn't happen, then the ants do not cross the water, and no food is found. If it does happen, then the ants cross the water within 40 turns. This accounts for the ants solving the problem in 162 turns.

I tried this solutin three times with the display program. Twice, the ants didn't cross the water, but the third time they did. It is all a matter of luck in how many ants move to the bottom right.

None of the other solutions found after iteration 2500 act in this way.

12.5.3 Synopsis

The average fitness graph shows that 2500-3000 is a good number of iterations for this population size, but finding that new solution at iteration 4751 disproves this. Basically, I will keep to the 2500 iterations, and if a test looks like it could have found a better solution if allowed to run longer, I will re-run the test.

The fact that the best solution from test 10 works less than half of the time makes it debatable whether it is a good solution. By adding more ants to the problem, then there would automatically be more ants moving to the bottom right. If enough ants were added, then this solution would always succeed.

12.6 Heavy Food Problem (Test 11-12)

12.6.1 Input Parameters

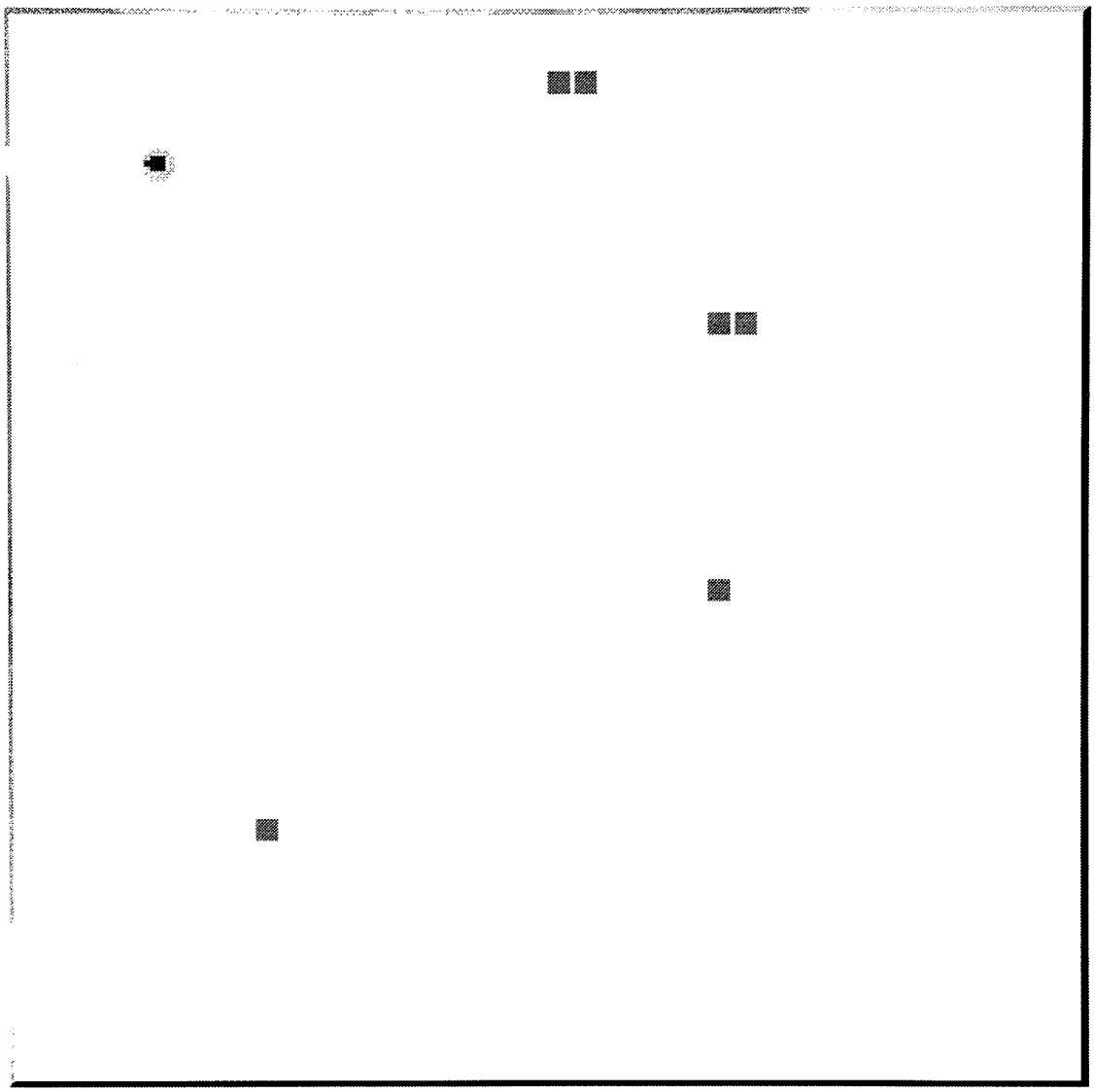
The parameters for tests 11 and 12 were:

Parameter	Tests 11	Test 12
POPULATION_SIZE	500	
TOURNAMENT_SIZE	4	
FOOD_LEFT_WEIGHT	10000000	
FOOD_UNFOUND_WEIGHT	100000	
TIME_WEIGHT	1	
NODE_WEIGHT	10000	
NODE_GROUP	50	
LEAF_ODDS	9	
NON_LEAF_ODDS	6	
MUTATE_ODDS	100	
PERCENT_GREEDY_MUTATE	80	
MAX_TURNS	1000	
NUM_ANTS	20	
GOAL_FITNESS	0	
POP_SEED	0 or 1	
INTERP_SEED	0	
PHEROMONE_STRENGTH	400	
PHEROMONE_SPREAD	40	
WAIT_ODDS	0	
AUTO_FOOD_DROP	1	
DEBUG	0	
SHOW_TIME	100	
DUMP	10000	
MAX_ITERATIONS	2500	
RESTORE_ITERATION	0	
RESTORE_VERSION	0	
RESTORE_AMOUNT	0	
INSTRUCTION_VERSION	21	23
MAP_VERSION	21	
VERSION	21-22	23-24

The *PHEROMONE_SPREAD* was increased to 40, so that other ants could detect the pheromones faster. In this problem, the food is in not in piles. Most of the pieces of food are not near each other. The pheromones are for asking other ants to help pick up a piece of food.

In tests 11 and 12, the number of ants is greater than the combined weight of all of the pieces of food. Technically, this means that the ants do not need to call for help, and there will be no deadlock problems where all the ants are waiting for help. This will not be the case in test 13.

The map for tests 11 and 12 is:



Going from left to right, top to bottom, the weights of the food are: 3, 3, 1, 1, 6, 3.

The instruction set for test 11 (file instruction21) is:

```
MOVE RANDOM
PICK UP FOOD
RELEASE PHEROMONE
PROC2
IF ( FOOD HERE ) THEN
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
IF ( MOVE TO NEST ) THEN
IF ( CANT LIFT FOOD ) THEN
```

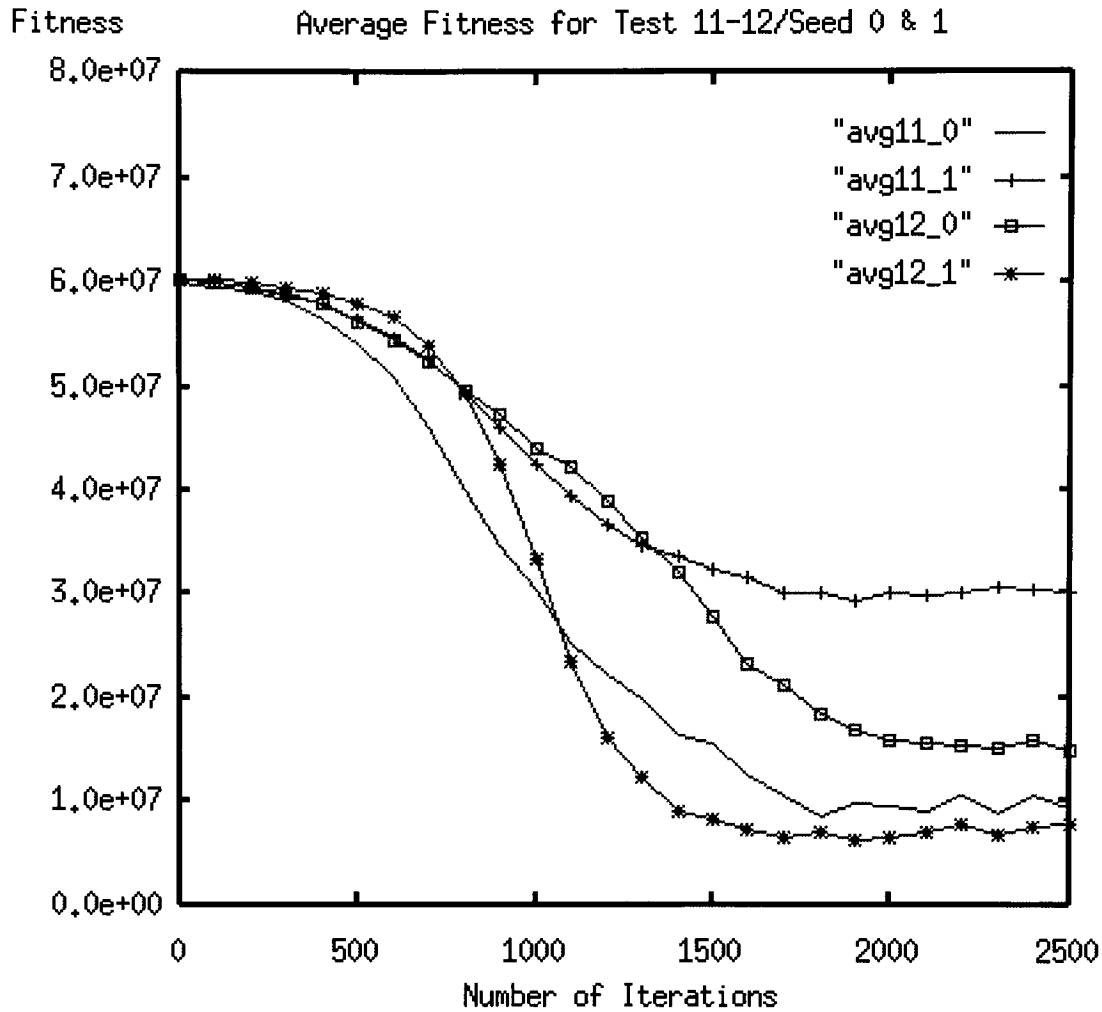
The instruction set for test 12 (file instruction23) is:

```
MOVE RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE
PROC2
IF ( FOOD HERE ) THEN
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
IF ( CANT LIFT FOOD ) THEN
```

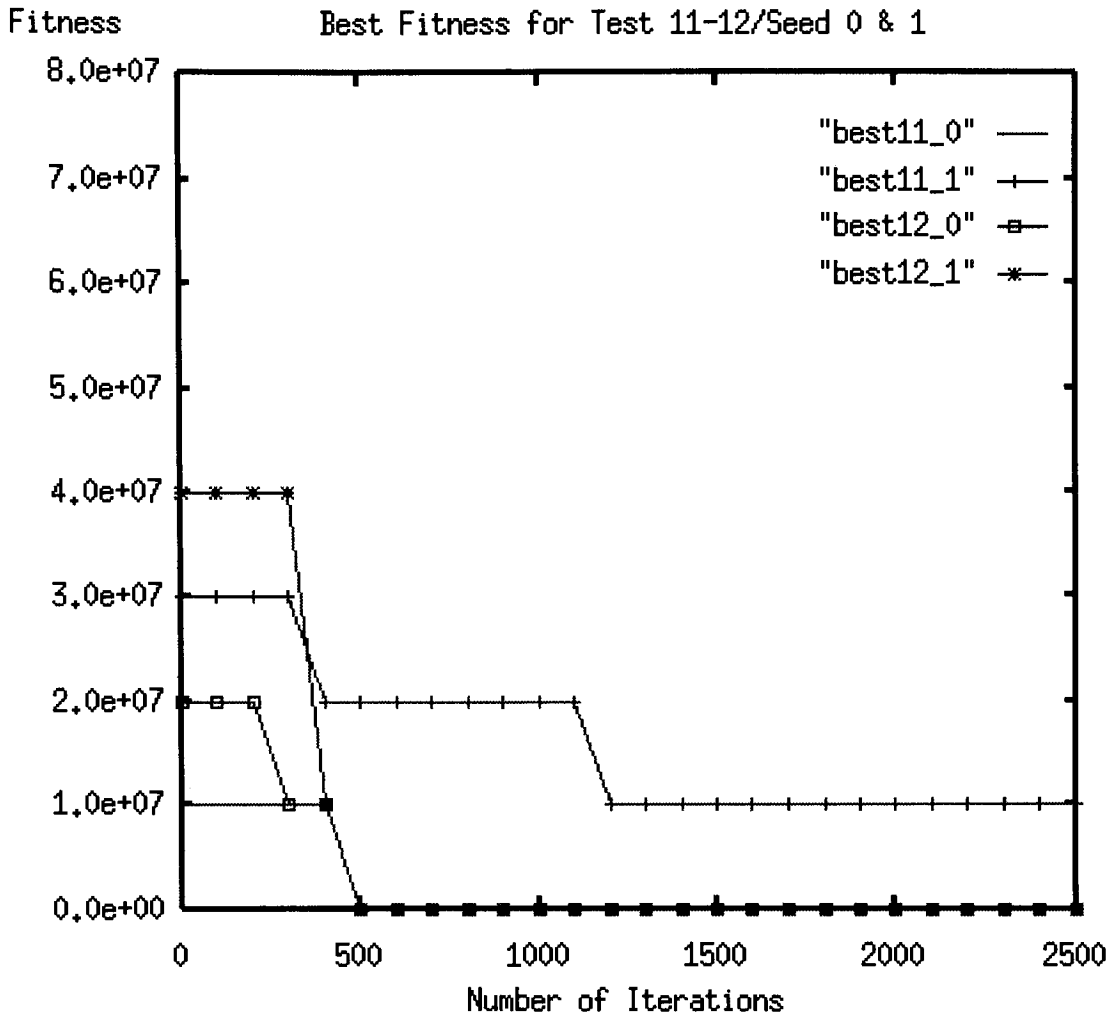
The difference between instruction set 21 and 23 is that the IF (MOVE TO NEST) THEN was replaced with MOVE TO NEST.

12.6.2 Output Data

The fitness plots for test 11 and 12 are:



The average fitness graph shows that the minor change in the instruction set did not make much of a difference. Maybe it could be said that test 12 was slightly better, because test 11 did bad on one of the seeds. Test 12 was the one that used MOVE TO NEST without the IF.



Again, the one seed in test 11 did badly, but the other seeds were about the same.

Test 11 Heavy Food with IF (MOVE TO NEST) - seed 0

```
Iteration =  2228, Time = Thu Mar 20 20:26:09, Fitness =     330,
     Number of Nodes =  21, Number of Turns =  330,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( CARRYING FOOD ) THEN
    IF ( FOOD HERE ) THEN
      IF ( MOVE TO NEST ) THEN
        PICK UP FOOD
      ELSE
        IF ( MOVE TO NEST ) THEN
          IF ( CANT LIFT FOOD ) THEN
            PICK UP FOOD
          ELSE
            PICK UP FOOD
        ELSE
          PICK UP FOOD
          MOVE RANDOM
          RELEASE PHEROMONE
    ELSE
      IF ( MOVE TO NEST ) THEN
        PICK UP FOOD
      ELSE
        RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      MOVE RANDOM
```

Note The IF (CARRYING FOOD) statement is true if the ant is trying to lift food, but not enough ants are helping.

Note The IF (MOVE TO ADJACENT FOOD) statement is true if food is adjacent, even if the ant cannot move.

This program does almost what you would expect the best solution to do. The ants look for food. When they find food, and can't pick it up, they release pheromones until more ants show up to help carry the food. Then all of the ants carry the food back to the nest.

The above does not happen if two pieces of food are next to each other. When the ant can't pick up the food, and food is adjacent, it just does another pick up food. When there are enough ants to pick up the food, it will move to the nest, but the next turn it will do a move to adjacent food. It will move 1 to the nest, and then back to the food, until other ants pick up the other food. Ants at piles of more than 1 piece of food will never release pheromones.

Test 11 Heavy Food with IF (MOVE TO NEST) - seed 1

```
Iteration =  1150, Time = Thu Mar 20 19:28:10, Fitness =  10001000,
     Number of Nodes =  15, Number of Turns = 1000,
     Food Left =  1, Food Not Found =  0
IF ( FOOD HERE ) THEN
  PICK UP FOOD
ELSE
  IF ( CANT LIFT FOOD ) THEN
    PICK UP FOOD
  ELSE
    IF ( CARRYING FOOD ) THEN
      IF ( MOVE TO NEST ) THEN
        IF ( MOVE TO NEST ) THEN
          IF ( CANT LIFT FOOD ) THEN
            PICK UP FOOD
          ELSE
            PICK UP FOOD
        ELSE
          RELEASE PHEROMONE
      ELSE
        MOVE RANDOM
    ELSE
      IF ( FOOD HERE ) THEN
        PICK UP FOOD
      ELSE
        MOVE RANDOM
```

This is the seed test that performed poorly. This program does not look for adjacent food. The ants have to stumble right on the piece of food. The ants also do not use pheromones to call for help.

Test 12 - Heavy Food with MOVE TO NEST seed 0

```
Iteration =  1864, Time = Thu Mar 20 19:35:27, Fitness =     504,
     Number of Nodes =  7, Number of Turns =  504,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( FOOD HERE ) THEN
    PICK UP FOOD
  ELSE
    IF ( CARRYING FOOD ) THEN
      MOVE TO NEST
    ELSE
      MOVE RANDOM
```

This program also does not use pheromones, but it fared better than test 11 seed 1, because it looks for adjacent food. It took 504 turns for the ants to find all of the food with-ough pheromones. Test 11 seed 0 took only 330 turns, because it did use pheromones.

Test 12 Heavy Food with MOVE TO NEST seed 1

```
Iteration =  1656, Time = Thu Mar 20 19:07:09, Fitness =     429,
    Number of Nodes =  15, Number of Turns =  429,
    Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  IF ( FOOD HERE ) THEN
    PICK UP FOOD
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      MOVE TO NEST
    ELSE
      MOVE RANDOM
ELSE
  PICK UP FOOD
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
```

This program only releases pheromones when it does not detect pheromones and it cannot pick up food. This can prevent the pheromone clouds from getting too big. When they get too big, the cloud will be around long after the food is gone. Still, this solution did not do as well as test 11 seed 0. It appears having the ants find the food faster is more important than not having ants get lost in a pheromone cloud with no food. This may change when the number of ants is lowered.

The problem with this program is that ants that are carrying food back to the nest will follow any pheromones that they smell. Following pheromones has a higher precedent than moving to the nest with food. Otherwise, this solution is a perfect solution. Except for the pheromones, it does not suffer any problems from having two pieces of food be adjacent.

12.6.3 Synopsis

It appears that the best solution is

```
IF ( CARRYING FOOD ) THEN
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    MOVE TO NEST
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    IF ( FOOD HERE ) THEN
      PICK UP FOOD
    ELSE
      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
        PICK UP FOOD
      ELSE
        MOVE RANDOM
```

The only unnecessary statement in the instruction set is the PROC2 statement. If the MOVE TO NEST is replaced with IF (MOVE TO NEST), then the IF (CANT LIFT FOOD) statement is also unnecessary.

12.7 Heavy Food Problem (Test 13)

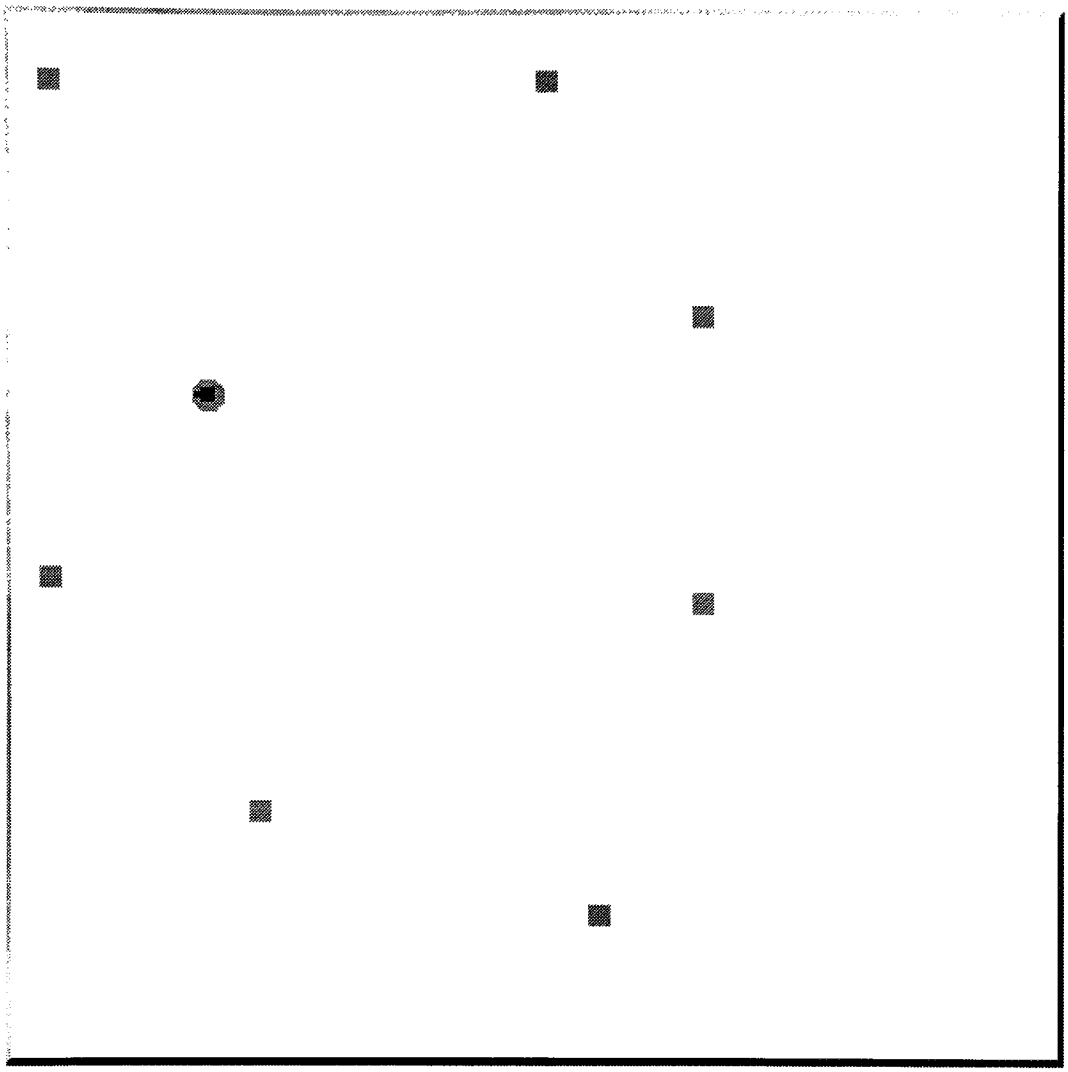
12.7.1 Input Parameters

In this test (test 13), the number of ants was decreased, and the amount of food was increased. This was done to force the ants to solve the deadlock problem that can occur when all of the ants are waiting for help to lift up food. To help the ants solve this problem, the pheromone parameters were increased, and the ants were given more time to solve the problem. The *WAIT_ODDS* parameter is used in the IF (TIRED OF WAITING) THEN statement.

The parameters for test 13 compared with the previous tests were:

Parameter	Tests 11 &12	Test 13
POPULATION_SIZE	500	
TOURNAMENT_SIZE	4	
FOOD_LEFT_WEIGHT	10000000	
FOOD_UNFOUND_WEIGHT	100000	
TIME_WEIGHT	1	
NODE_WEIGHT	10000	
NODE_GROUP	50	
LEAF_ODDS	9	
NON_LEAF_ODDS	6	
MUTATE_ODDS	100	
PERCENT_GREEDY_MUTATE	80	
MAX_TURNS	1000	1500
NUM_ANTS	20	15
GOAL_FITNESS	0	
POP_SEED	0 or 1	
INTERP_SEED	0	
PHEROMONE_STRENGTH	400	500
PHEROMONE_SPREAD	40	50
WAIT_ODDS	0	50
AUTO_FOOD_DROP	1	
DEBUG	0	
SHOW_TIME	100	
DUMP	10000	
MAX_ITERATIONS	2500	
RESTORE_ITERATION	0	
RESTORE_VERSION	0	
RESTORE_AMOUNT	0	
INSTRUCTION_VERSION	21,23	25
MAP_VERSION	21	25
VERSION	21-24	25-26

The map file used is:



All of the food in the map has a weight of 6.

The instruction set was:

```
MOVE QUASI RANDOM
PICK UP FOOD
RELEASE PHEROMONE
ESCAPE PHEROMONE
PROC2
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
IF ( MOVE TO NEST ) THEN
IF ( TIRED OF WAITING ) THEN
```

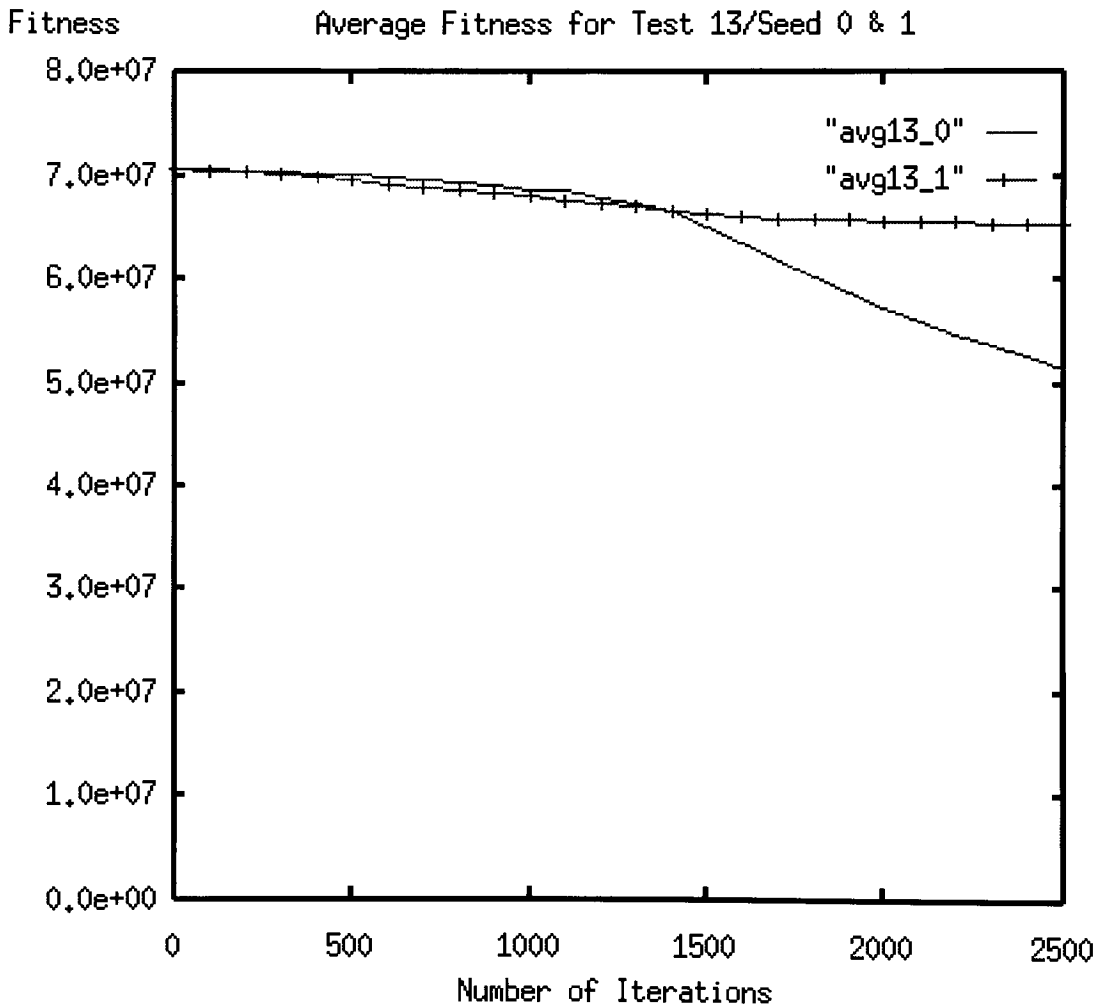
Extra statements were added to fix the deadlock problem. These statements were:

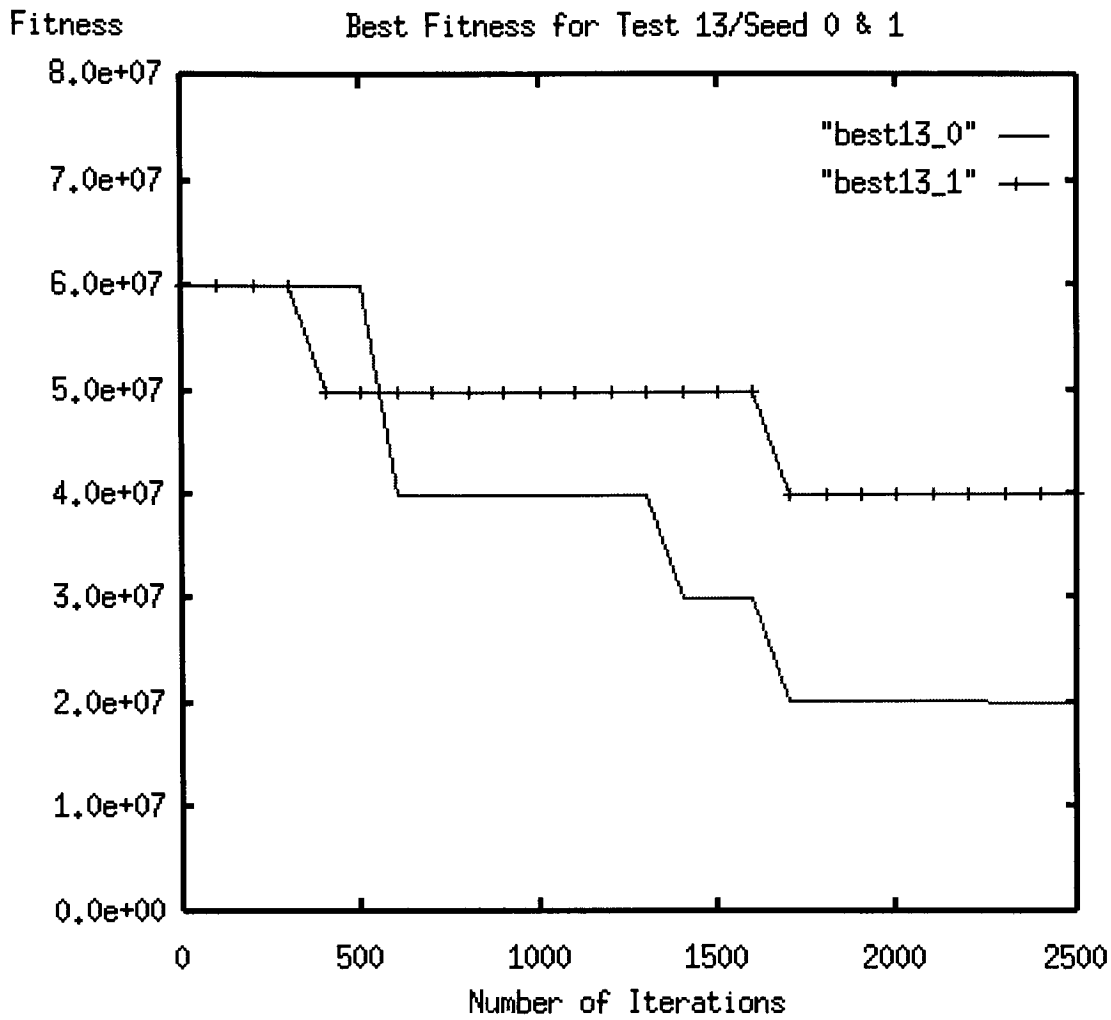
```
IF ( TIRED OF WAITING ) THEN
ESCAPE PHEROMONE
```

These statements allow the ants a random method of leaving one piece of food, and going to help other ants lift a different piece of food. The IF (TIRED OF WAITING) THEN statement has a 1 in *WAIT_ODDS* chance of being true. The ESCAPE PHEROMONE statement has the ant drop its food and move forward until it is outside the pheromone cloud, or has hit the edge of the map. This allows the ant to find a different pheromone cloud around a different piece of food.

12.7.2 Output Data

The fitness graphs for this test are:





None of the solutions in this test succeeded in bringing all of the food back to the nest. The second seed was much worse, only bringing back 3 pieces of food. Also, with the second seed, the average fitness did not change much after 2500 iterations.

The best solutions for both seeds are:

Test 13 Heavy Food With Deadlock - seed 0

```
Iteration =  2242, Time = Sat Mar 22 19:41:43, Fitness =  20001500,
     Number of Nodes =  29, Number of Turns = 1500,
     Food Left =  2, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  RELEASE PHEROMONE
ELSE
  IF ( TIRED OF WAITING ) THEN
    RELEASE PHEROMONE
  ELSE
```

```
IF ( TIRED OF WAITING ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE QUASI RANDOM
  ELSE
    MOVE QUASI RANDOM
ELSE
  IF ( MOVE TO ADJACENT PHEROMONE ) THEN
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      ESCAPE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      MOVE QUASI RANDOM
    ELSE
      MOVE QUASI RANDOM
      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
        IF ( MOVE TO ADJACENT PHEROMONE ) THEN
          RELEASE PHEROMONE
        ELSE
          RELEASE PHEROMONE
      ELSE
        MOVE QUASI RANDOM
      PICK UP FOOD
  PICK UP FOOD
IF ( TIRED OF WAITING ) THEN
  PICK UP FOOD
ELSE
  PICK UP FOOD
```

If given more turns to execute, this may be a good solution. In this solution, ants will release pheromones:

- When they move to adjacent food.
- When they are tired of waiting.
- When they move to pheromones, and after the move there are no pheromones adjacent

Thus, ants will not release pheromones when they are at food and pheromones are present. There will not be large pheromone clouds around the food. Probably just a 3 block wide cloud. The ESCAPE PHEROMONE statement is not required to leave the food. Only a MOVE QUASI RANDOM is required. Also, after the food is removed, there is no large pheromone cloud to attract other ants.

The problem with this solution is that it does not use IF (MOVE TO NEST). So it must randomly find the nest. Also, pheromone clouds are not used to attract other ants. Which piece of food an ant will find next is purely random. Still, with a pure random solution, the ants will eventually solve the problem. And, the solution does get around the deadlock problem.

Test 13 - Heavy Food With Deadlock - seed 1

```
Iteration =  1660, Time = Sat Mar 22 19:34:36, Fitness =  40001500,
    Number of Nodes =  35, Number of Turns = 1500,
    Food Left =  4, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
  PICK UP FOOD
  PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO ADJACENT FOOD ) THEN
      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
        PICK UP FOOD
      ELSE
        PICK UP FOOD
    ELSE
      IF ( MOVE TO ADJACENT FOOD ) THEN
        IF ( MOVE TO NEST ) THEN
          MOVE QUASI RANDOM
        ELSE
          RELEASE PHEROMONE
      ELSE
        IF ( MOVE TO ADJACENT PHEROMONE ) THEN
          PICK UP FOOD
        ELSE
          IF ( MOVE TO ADJACENT PHEROMONE ) THEN
            PICK UP FOOD
          ELSE
            IF ( MOVE TO ADJACENT PHEROMONE ) THEN
              IF ( MOVE TO ADJACENT FOOD ) THEN
                ESCAPE PHEROMONE
              ELSE
                PICK UP FOOD
            ELSE
              IF ( MOVE TO ADJACENT FOOD ) THEN
                IF ( MOVE TO ADJACENT FOOD ) THEN
                  PICK UP FOOD
                ELSE
                  RELEASE PHEROMONE
              ELSE
                IF ( MOVE TO ADJACENT FOOD ) THEN
                  IF ( CARRYING FOOD ) THEN
                    RELEASE PHEROMONE
                  ELSE
                    MOVE QUASI RANDOM
                ELSE
                  RELEASE PHEROMONE
    PICK UP FOOD
  ELSE
    MOVE QUASI RANDOM
```

This solution does almost nothing. Most of the code will never be executed, because two pieces of food would have to be next to each other. With useless statements removed, the solution is:

```
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  MOVE QUASI RANDOM
```

Yet, this solution still managed to bring 3 pieces of food back to the nest. Although this is after testing this solution many times.

12.7.3 Synopsis

Neither of these tests used ESCAPE PHEROMONE. In fact, they hardly used pheromones at all to call for help. If the ants were to use pheromones, then wandering ants would be more likely to find the biggest pheromone cloud, which would be the one with the most ants.

If the food was placed closer together, and the pheromone clouds were closer, or even overlapped, than the ants might move from one piece of food to the next.

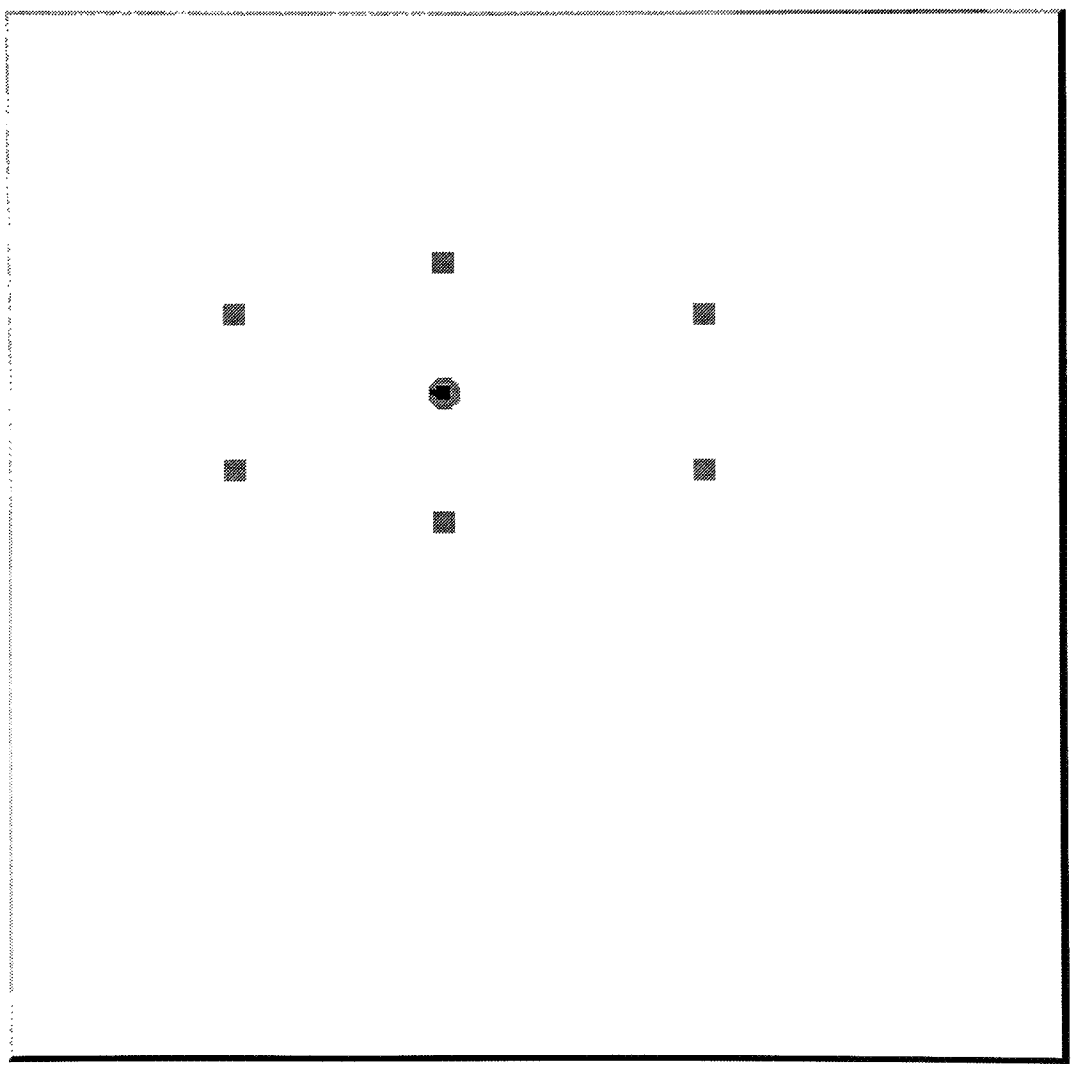
This same effect could be created by increasing *PHEROMONE_STRENGTH*, but that would also cause the solutions to take longer to interpret.

12.8 Heavy Food Problem (Test 14)

12.8.1 Input Parameters

The instruction set and parameter files for test 14 were the basically same as test 13. The only differences in the parameter files were the *VESRION* and *MAP_VERSION* parameters.

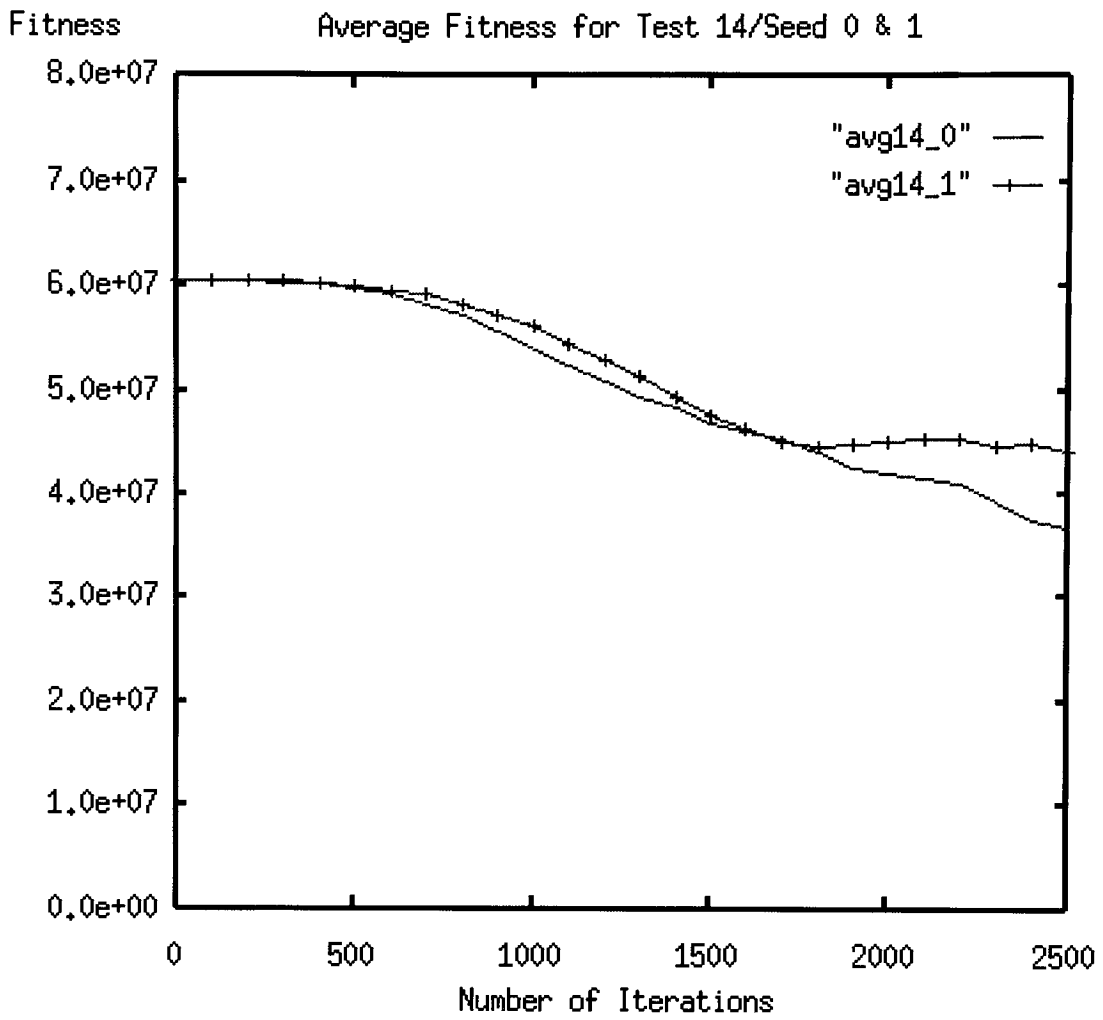
This test is an attempt to get the ants to use pheromones more to call for help by putting the food closer together. The map file for test 14 is:



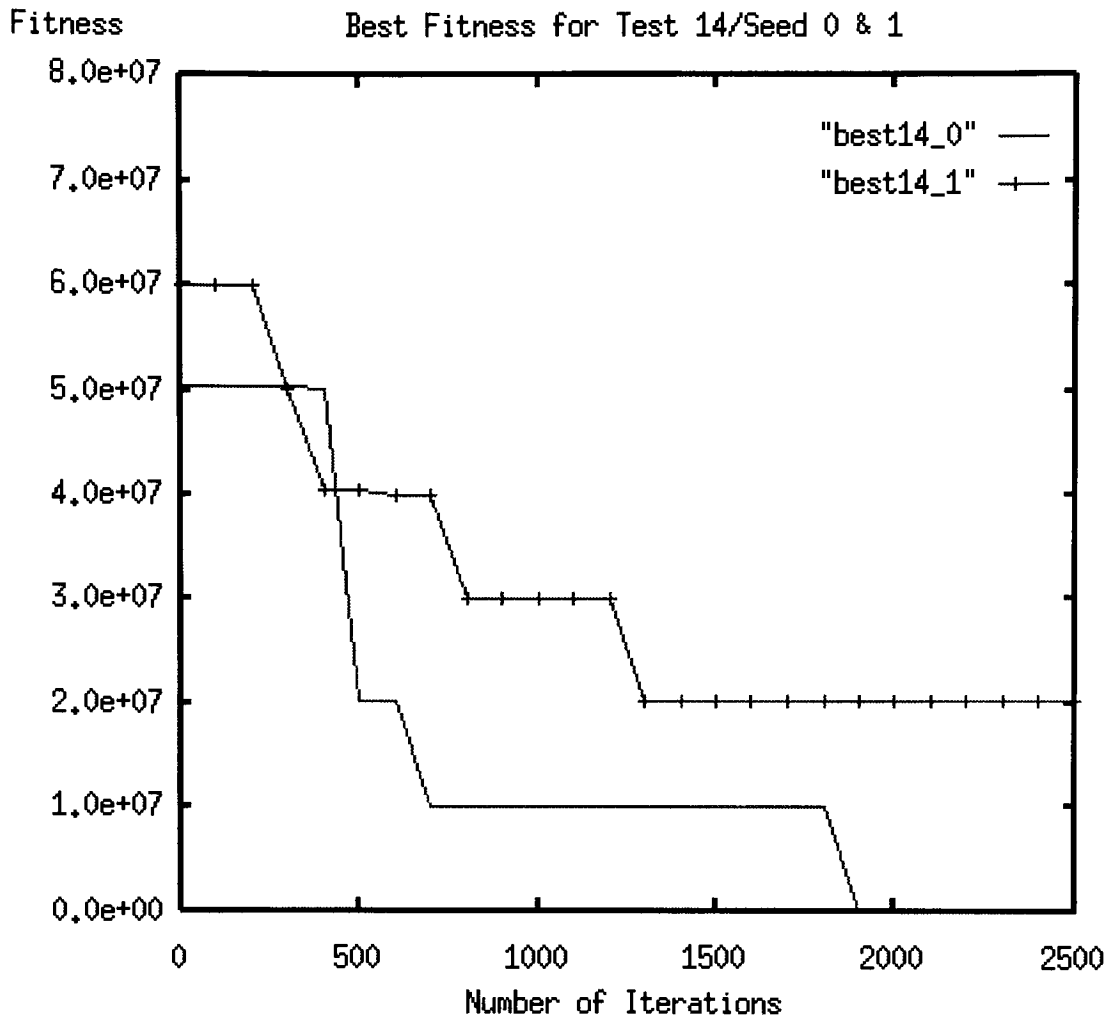
Each piece of food in the map has a weight of 6.

12.8.2 Output Data

The fitness graphs for this test are:



This test appears to perform better than the previous test. This could just be a result of having all of the food closer to the nest. It will take less turns to find the food..



Again, both seeds are better than the previous test. The first seed found a solution that got all of the food back to the nest.

The first seed continued to find better solutions (found all of the food in less turns), while the second seed stopped finding any better solutions after iteration 1222.

The best solutions are:

Test 14 - Close Heavy Food with Deadlock - seed 0

```
Iteration =  2372, Time = Tue Mar 25 00:10:04, Fitness =    1259,
     Number of Nodes =  17, Number of Turns = 1259,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT FOOD ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    IF ( MOVE TO NEST ) THEN
      IF ( CARRYING FOOD ) THEN
        PICK UP FOOD
      ELSE
        ESCAPE PHEROMONE
        PICK UP FOOD
    ELSE
      ESCAPE PHEROMONE
  ELSE
    PICK UP FOOD
ELSE
  MOVE QUASI RANDOM
RELEASE PHEROMONE
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  MOVE QUASI RANDOM
ELSE
  RELEASE PHEROMONE
```

Half of the statements in this first solution are unused. This solution does not use the ESCAPE PHEROMONE or IF (TIRED OF WAITING) statements. Still, with this solution, the ants found all of the food.

The reason the first solution works, is that pheromones are released by every ant on every turn, and the ants will follow the strongest pheromones. Thus, all of the ants will basically be moving as a group. When they stay together, then they will all be around when food is found. Still, there are two MOVE QUASI RANDOM statements executed for every MOVE TO ADJACENT PHEROMONE, so the ants will not always be right next to each other. Eventually, they would be able to spread apart.

Trying this solution on the map from test 13 where the food was spread out yielded bad results. The ants are no longer together when the food is found.The problem could also be that the ants do not use the MOVE TO NEST statement, and must find the nest randomly. This is harder when the food is not near the nest. Still, this is a good solution when the ants are working in a small area.

Test 14 - Close Heavy Food with Deadlock - seed 1

```
Iteration =  1222, Time = Mon Mar 24 20:44:30, Fitness =  20101500,
     Number of Nodes =  17, Number of Turns = 1500,
     Food Left =  2, Food Not Found =  1
IF ( MOVE TO NEST ) THEN
  IF ( MOVE TO ADJACENT FOOD ) THEN
    PICK UP FOOD
  ELSE
    IF ( MOVE TO NEST ) THEN
      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
        MOVE QUASI RANDOM
      ELSE
        IF ( MOVE TO ADJACENT FOOD ) THEN
          MOVE QUASI RANDOM
        ELSE
          RELEASE PHEROMONE
        ESCAPE PHEROMONE
    ELSE
      IF ( TIRED OF WAITING ) THEN
        MOVE QUASI RANDOM
      ELSE
        ESCAPE PHEROMONE
    MOVE QUASI RANDOM
ELSE
  RELEASE PHEROMONE
```

This second solution finds some of the food well. Again, the ants are constantly releasing pheromones when they dont detect any. This means that when they start, they all release a pheromone. Then, they all stay within the pheromones until they dissipate, or an any finds food, and starts releasing more pheromones.

This solution suffers from the problem of too large pheromone clouds. They ants are unlikely to find food outside the pheromone cloud until it dissipates. There is also a lot of wasted movement in this solution. The ants are constantly moving randomly. Each move random is a turn for that ant, so the this solution needs more time (turns) to solve the problem.

Still, this soltution does solve the deadlock problem. It also knows to bring the food directly to the nest. The other problem with this solution is the fact that it does a MOVE TO NEST every turn. This prevents the ants from getting far from the nest. If given enough time, the ants can find all of the food except the food to the right of the nest, because that food is the furthest from the nest.

12.8.3 Synopsis

Technically, test 14 seed 0 did find a solution to the given problem. Unfortunately, this solution was limited to the given map where all of the food was close together. The solution did not work for other maps where the food was spread apart. Thus, it is not an acceptable solution.

The recent tests appear to have problems moving to the nest after they have found the food. Therefore, the MOVE TO NEST statement, not as an if statement, should be added back to the instruction set.

The reason for putting the food closer together in the first place was to have pheromone clouds overlap. Instead, the Genetic Program found another way to solve the problem, taking advantage of the fact that the food was near the nest. By increasing the *PHEROMONE_SPREAD* parameter, the pheromone clouds will increase in size, which will allow them to be closer together. Also, the pheromone clouds will dissipate faster.

12.9 Heavy Food Problem (Test 15-16)

12.9.1 Input Parameters

These last tests attempt to use the population set from test 12 seed 1 as a starting point to solve the heavy food problem. Test 12 seed 1 was a near perfect solution for a simple heavy food problem, where there are more ants than the combined weight of the food. All 500 programs in the population from this test were combined with 500 new solutions with a slightly different instruction set, to create a starting population of 1000 programs.

The parameters for test 15 and 16 compared with the previous test 13 were:

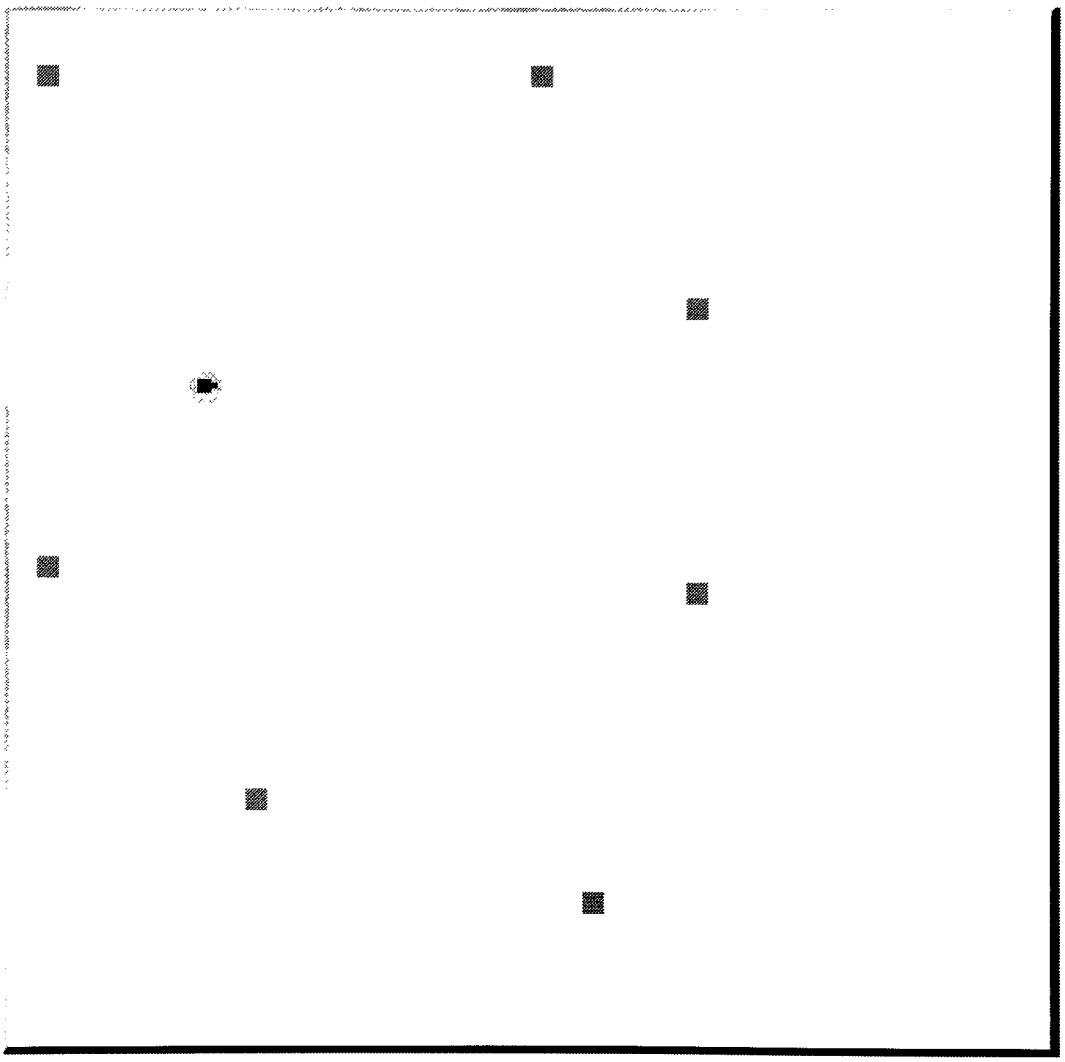
Parameter	Tests 13	Test 15	Test16
POPULATION_SIZE	500	1000	1000
TOURNAMENT_SIZE	4		
FOOD_LEFT_WEIGHT	10000000		
FOOD_UNFOUND_WEIGHT	100000		
TIME_WEIGHT	1		
NODE_WEIGHT	10000		
NODE_GROUP	50		
LEAF_ODDS	9		
NON_LEAF_ODDS	6		
MUTATE_ODDS	100		
PERCENT_GREEDY_MUTATE	80		
MAX_TURNS	1500	2500	2500
NUM_ANTS	15		
GOAL_FITNESS	0		
POP_SEED	0 or 1		
INTERP_SEED	0		
PHEROMONE_STRENGTH	500		
PHEROMONE_SPREAD	50	70	70
WAIT_ODDS	50	50	100
AUTO_FOOD_DROP	1		
DEBUG	0		
SHOW_TIME	100		
DUMP	10000		
MAX_ITERATIONS	2500		4000
RESTORE_ITERATION	0	2000	2000
RESTORE_VERSION	0	24	24
RESTORE_AMOUNT	0	500	500
INSTRUCTION_VERSION	25	29	29
MAP_VERSION	25	29	31
VERSION	25-26	29-30	31-32

The same instruction set was used for tests 15 and 16. The statements ESCAPE PHEROMONE and IF (TIRED OF WAITING) THEN were added from the original instruction set from test 12.

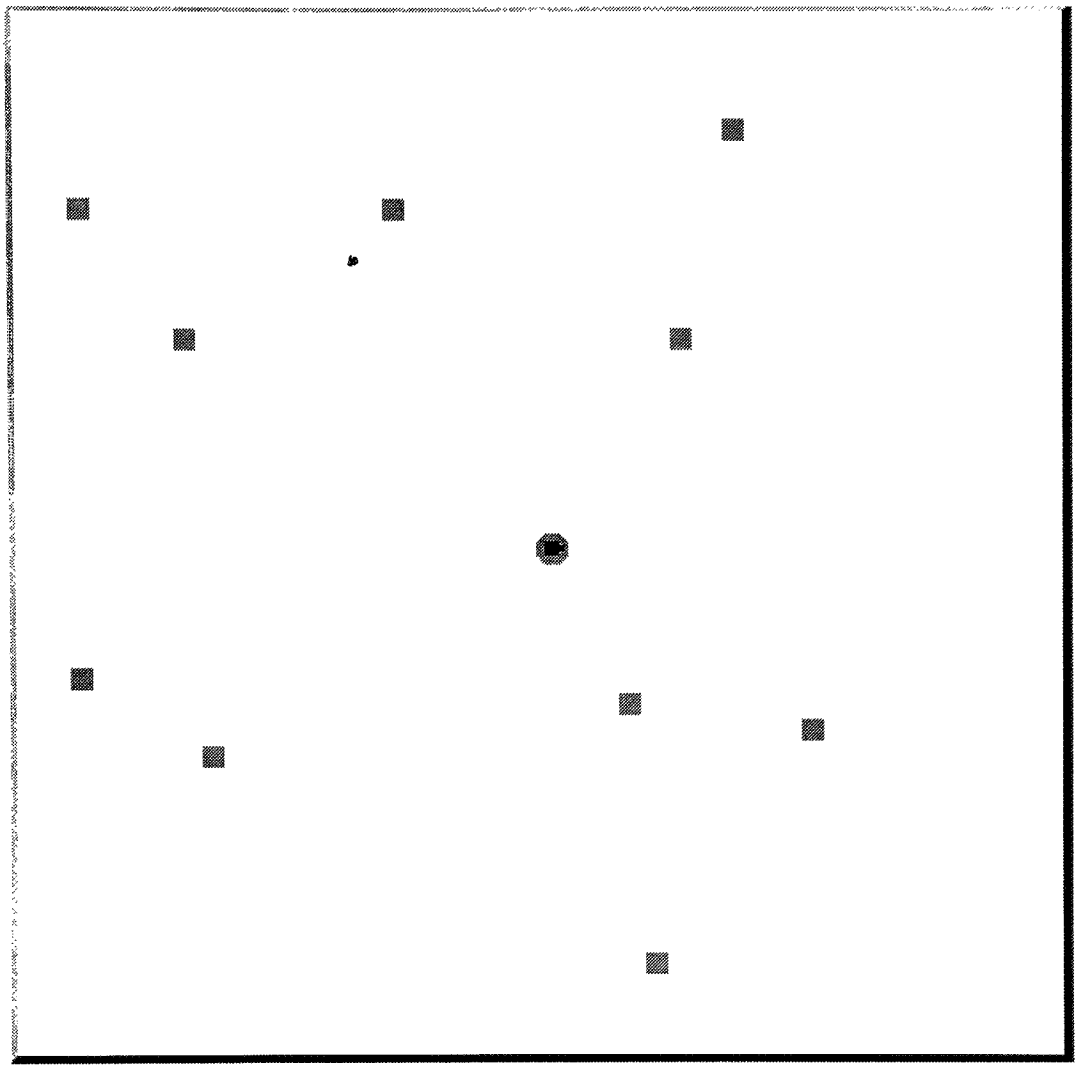
MOVE QUASI RANDOM
MOVE TO NEST
PICK UP FOOD
RELEASE PHEROMONE
ESCAPE PHEROMONE
PROC2

IF (CARRYING FOOD) THEN
IF (MOVE TO ADJACENT FOOD) THEN
IF (MOVE TO ADJACENT PHEROMONE) THEN
IF (MOVE TO NEST) THEN
IF (TIRED OF WAITING) THEN

The map for test 15 was:



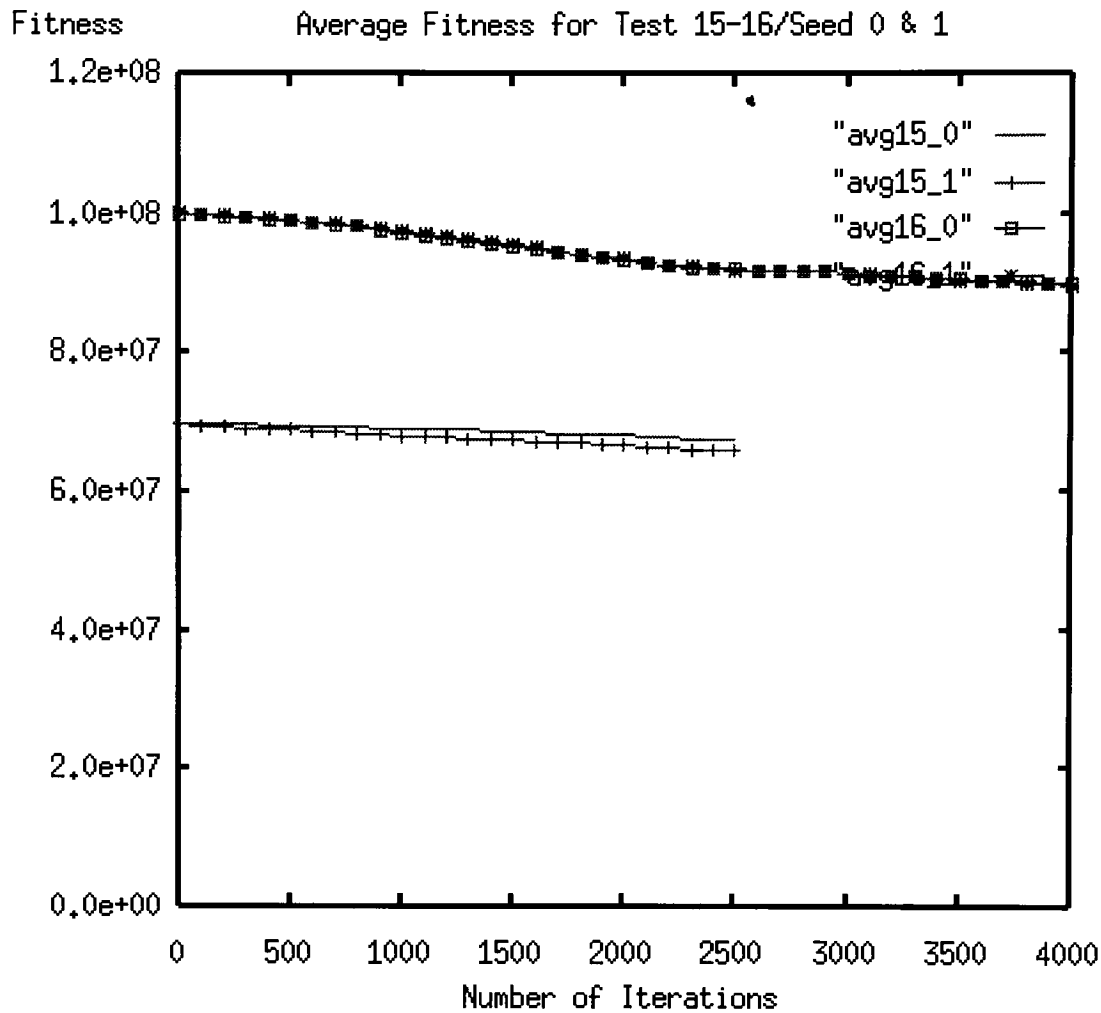
The map for test 16 was:



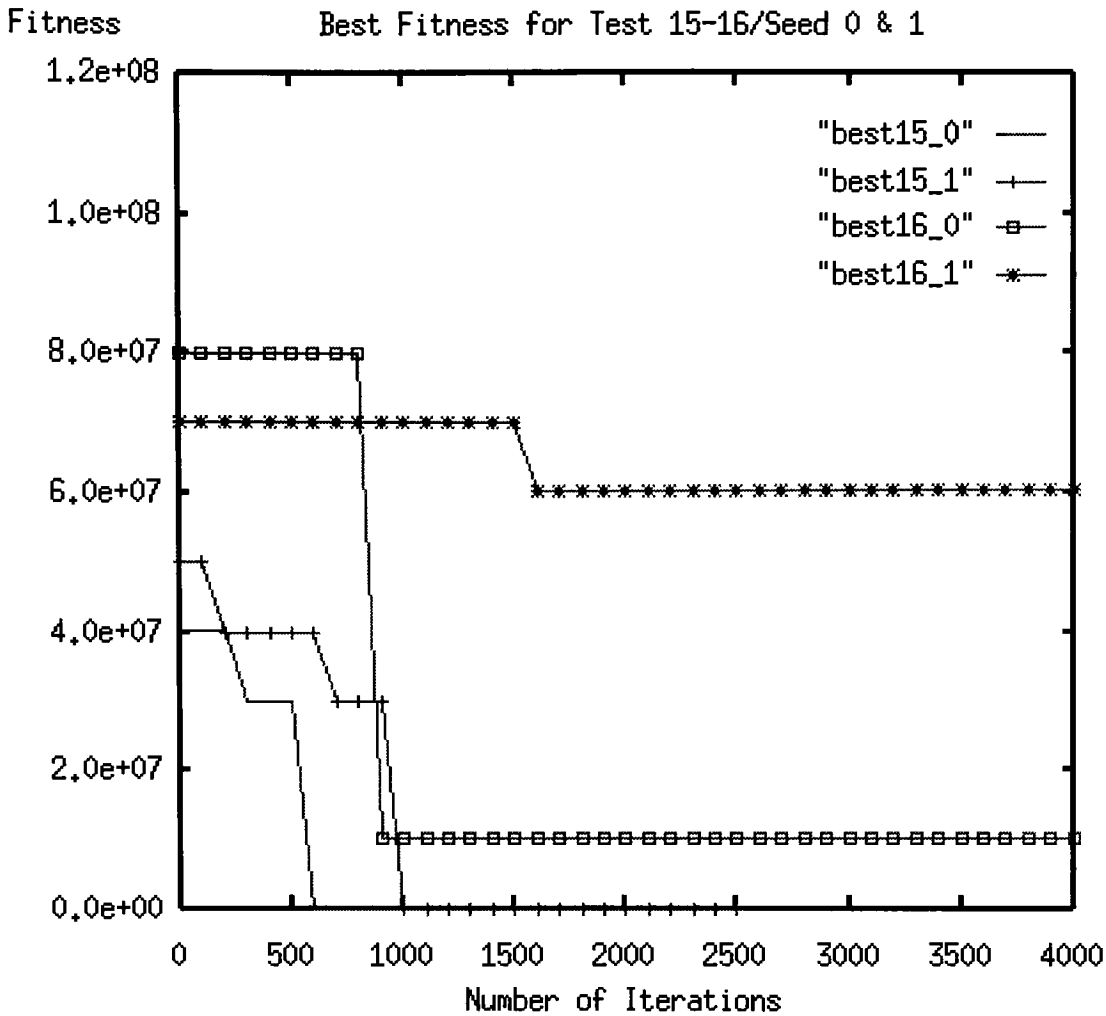
In both maps the weight of each piece of food is 9.

12.9.2 Output Data

Although test 16 was run for more iterations than test 15, both tests will be displayed in the same graph. Test 15 contained less food, so the fitnesses will always be less for this test. The importance of the graph is the slope of the fitness in how fast it decreases.



The average fitness for test 15 does not appear to change at all, while there is noticable change in test 16.



Both parts of test16 did not find a perfect solution. Test 15 found a perfect solution easily, although there was less food to find. When there is less food in the map, the chance that the ants will find all of the food by luck increases. They do not need to use a good program to find the food.

The best solutions found by these tests are.

Test 15 Heavy Food Problem Using Restore - Seed 0

```
Iteration =  536, Time = Thu Apr  3 20:07:27, Fitness =    2375,
     Number of Nodes =  15, Number of Turns = 2375,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  IF ( FOOD HERE ) THEN
    PICK UP FOOD
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      MOVE TO NEST
ELSE
  MOVE RANDOM
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
```

Test 15 - Heavy Food Problem Using Restore - Seed 1

```
Iteration =  925, Time = Thu Apr  3 21:11:25, Fitness =    1682,
     Number of Nodes =  15, Number of Turns = 1682,
     Food Left =  0, Food Not Found =  0
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  IF ( FOOD HERE ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      MOVE RANDOM
ELSE
  PICK UP FOOD
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
```

Test 16 - Heavy Food Problem Using Restore, Many Food- Seed 0

```
Iteration =  885, Time = Mon Apr 7 22:13:46, Fitness = 10102500,
    Number of Nodes = 15, Number of Turns = 2500,
    Food Left = 1, Food Not Found = 1
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  IF ( FOOD HERE ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      MOVE TO NEST
ELSE
  PICK UP FOOD
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      MOVE RANDOM
    ELSE
      IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
```

Test 16 - Heavy Food Problem Using Restore, Many Food- Seed 1

```
Iteration =  1595, Time = Mon Apr 7 23:50:01, Fitness = 60202500,
    Number of Nodes = 15, Number of Turns = 2500,
    Food Left = 6, Food Not Found = 2
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
  IF ( FOOD HERE ) THEN
    PICK UP FOOD
  ELSE
    IF ( MOVE TO ADJACENT PHEROMONE ) THEN
      PICK UP FOOD
    ELSE
      MOVE RANDOM
ELSE
  PICK UP FOOD
  IF ( CANT LIFT FOOD ) THEN
    RELEASE PHEROMONE
  ELSE
    IF ( MOVE TO ADJACENT FOOD ) THEN
      PICK UP FOOD
    ELSE
      IF ( CARRYING FOOD ) THEN
        MOVE TO NEST
      ELSE
        MOVE RANDOM
```

All four of these solutions are virtually identical. Except for minor differences, which probably are due to mutation, they are the same. These four solutions are also very similar to the best solution from test 14 seed 1, which is the test that was to restore the population from.

12.9.3 Synopsis

Test 12 was used to find a solution to the heavy food problem where the number of ants was greater than the combined weight of all of the food. By using the population of programs from this test at itteration, hopefully some of the information learned by this test would apply to a more complicated test. In this test, there could exist a deadlock condition, where all of the ants were waiting for help. Instead, the best solution from test 12 is also the best solution for the more complicated test. This was due in part to the fact that the ants could get lucky, and find all of the food without running into the deadlock condition. Also, because the solutions from the previous problem were so much better than the new random solutions, the new solutions were rarely used in crossover.

12.10 Water Crossing and Heavy Food Problem (Test 17)

12.10.1 Input Parameters

In test 17, there is a stream separating some of the food and the ants. Also, all of the food has a weight of 5. To help the genetic program get started, half of the population consists of the population of test 9 seed 0 at iteration 1500. Test 9 had the best results in the Water Crossing Problem.

There are 100 ants in this problem, so there will probably not be a deadlock problem with the heavy food. Many ants will die in so that the rest can cross the water. It is impossible to figure out the exact number that will allow the ants to cross the water, ant still leave enough alive to pick up the heavy food.

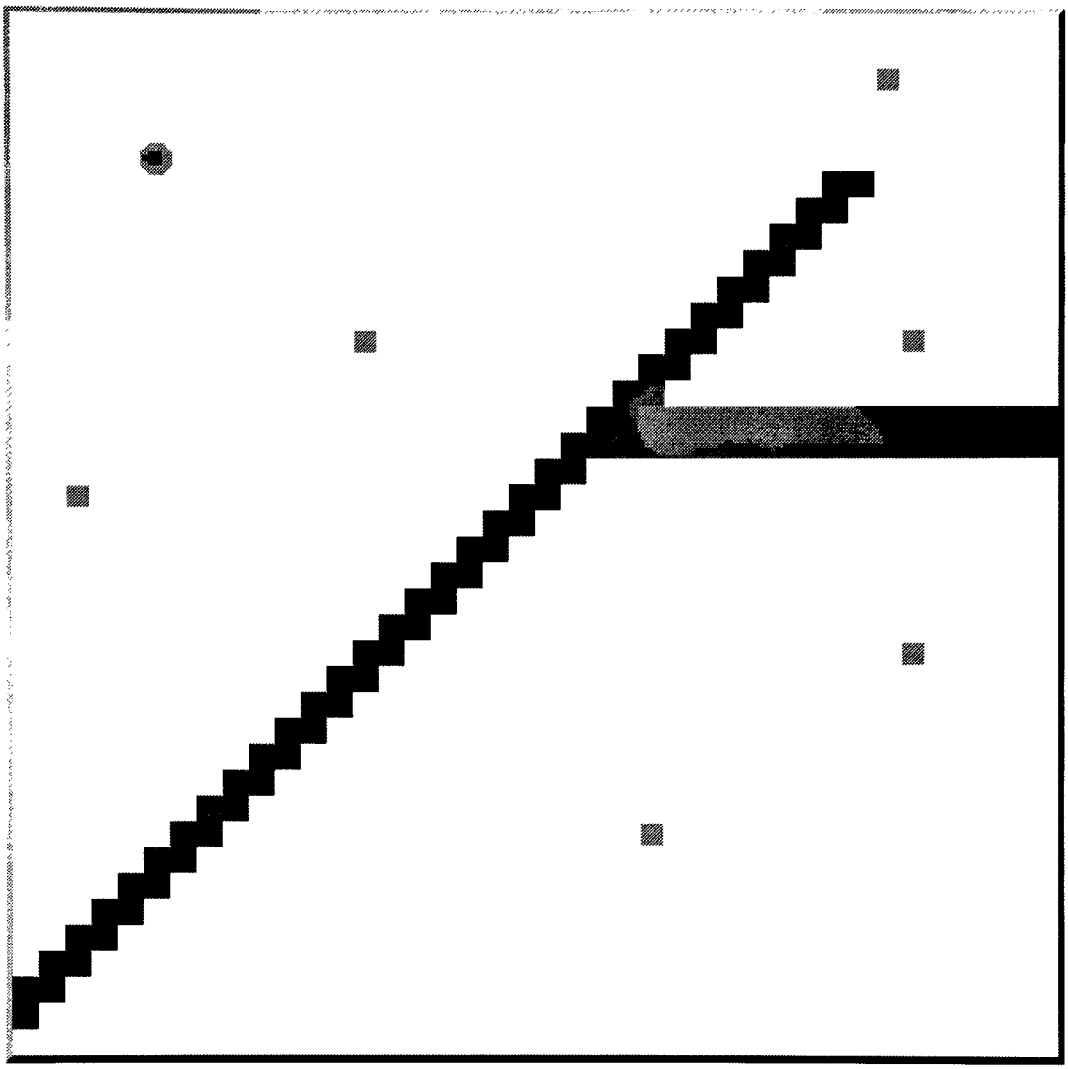
POPULATION_SIZE	1000
TOURNAMENT_SIZE	4
FOOD_LEFT_WEIGHT	10000000
FOOD_UNFOUND_WEIGHT	100000
TIME_WEIGHT	1
NODE_WEIGHT	10000
NODE_GROUP	50
LEAF_ODDS	9
NON_LEAF_ODDS	6
MUTATE_ODDS	100
PERCENT_GREEDY_MUTATE	80
MAX_TURNS	3000
NUM_ANTS	100
GOAL_FITNESS	0
POP_SEED	0
INTERP_SEED	0
PHEROMONE_STRENGTH	500
PHEROMONE_SPREAD	60
WAIT_ODDS	0
AUTO_FOOD_DROP	1
DEBUG	0
SHOW_TIME	100
DUMP	10000
MAX_ITERATIONS	2000
RESTORE_ITERATION	1500
RESTORE_VERSION	17
RESTORE_AMOUNT	500
INSTRUCTION_VERSION	33
MAP_VERSION	33
VERSION	33

The instruction set for the new solutions was rather large. It allowed the genetic program to solve the problem in many different ways. Basically, all of the statements for water crossing, plus those to solve the deadlock problem are necessary. The use of both move to pheromone statements allows the genetic program to decide if it is better to move to the strongest pheromone, on only the ones away from the nest.

```
MOVE QUASI RANDOM
MOVE TO NEST
MOVE TO NEST THROUGH WATER
PICK UP FOOD
RELEASE PHEROMONE
MOVE INTO WATER
ESCAPE PHEROMONE

PROC2
IF ( FOOD HERE ) THEN
IF ( CARRYING FOOD ) THEN
IF ( MOVE TO ADJACENT FOOD ) THEN
IF ( MOVE TO AWAY PHEROMONE ) THEN
IF ( MOVE TO DEAD ANT ) THEN
IF ( MOVE TO ADJACENT PHEROMONE ) THEN
IF ( MOVE TO NEST ) THEN
IF ( TIRED OF WAITING ) THEN
```

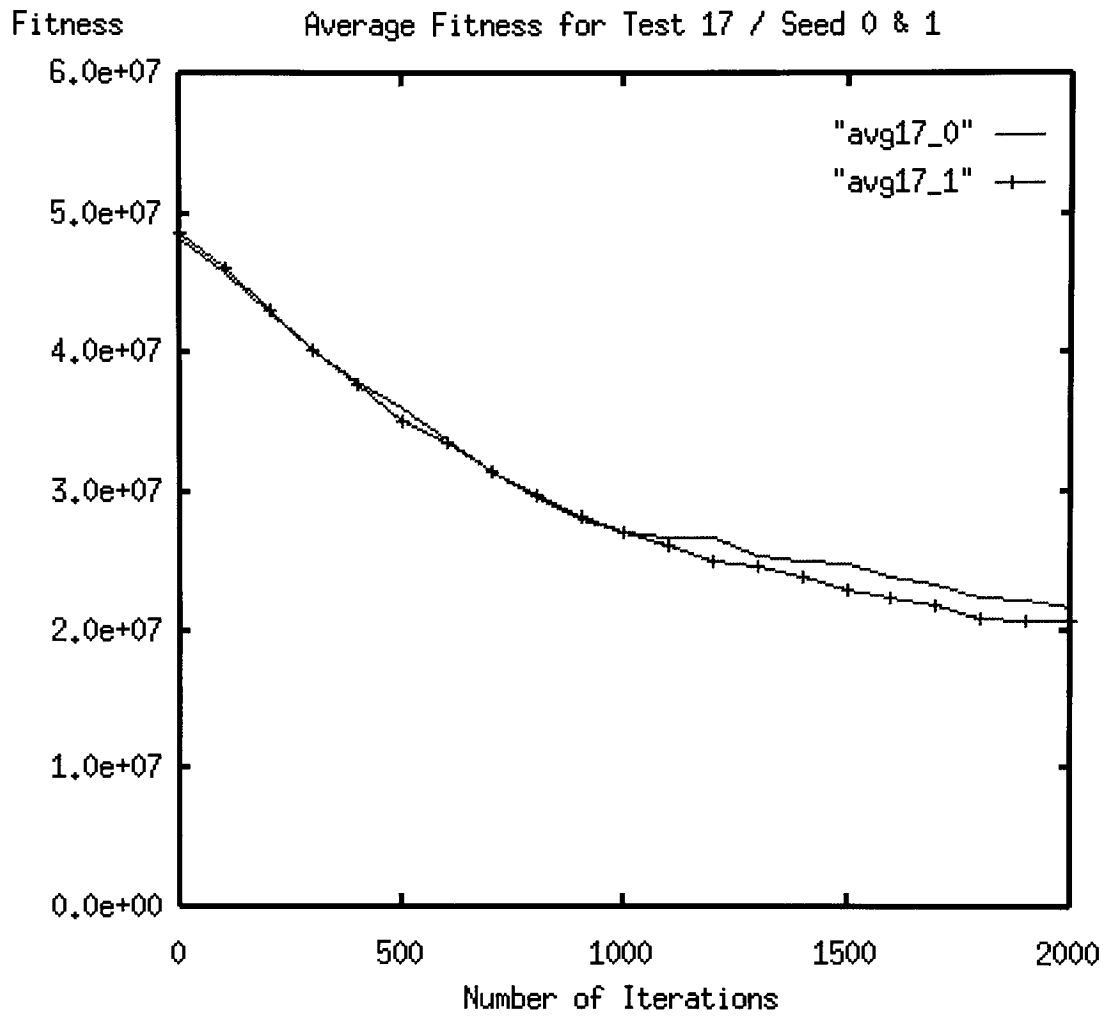
This is the map file used in this test. All of the food in this map has a weight of 5. The bar running from the bottom left to the top right is water. The horizontal bar is an impassable wall.



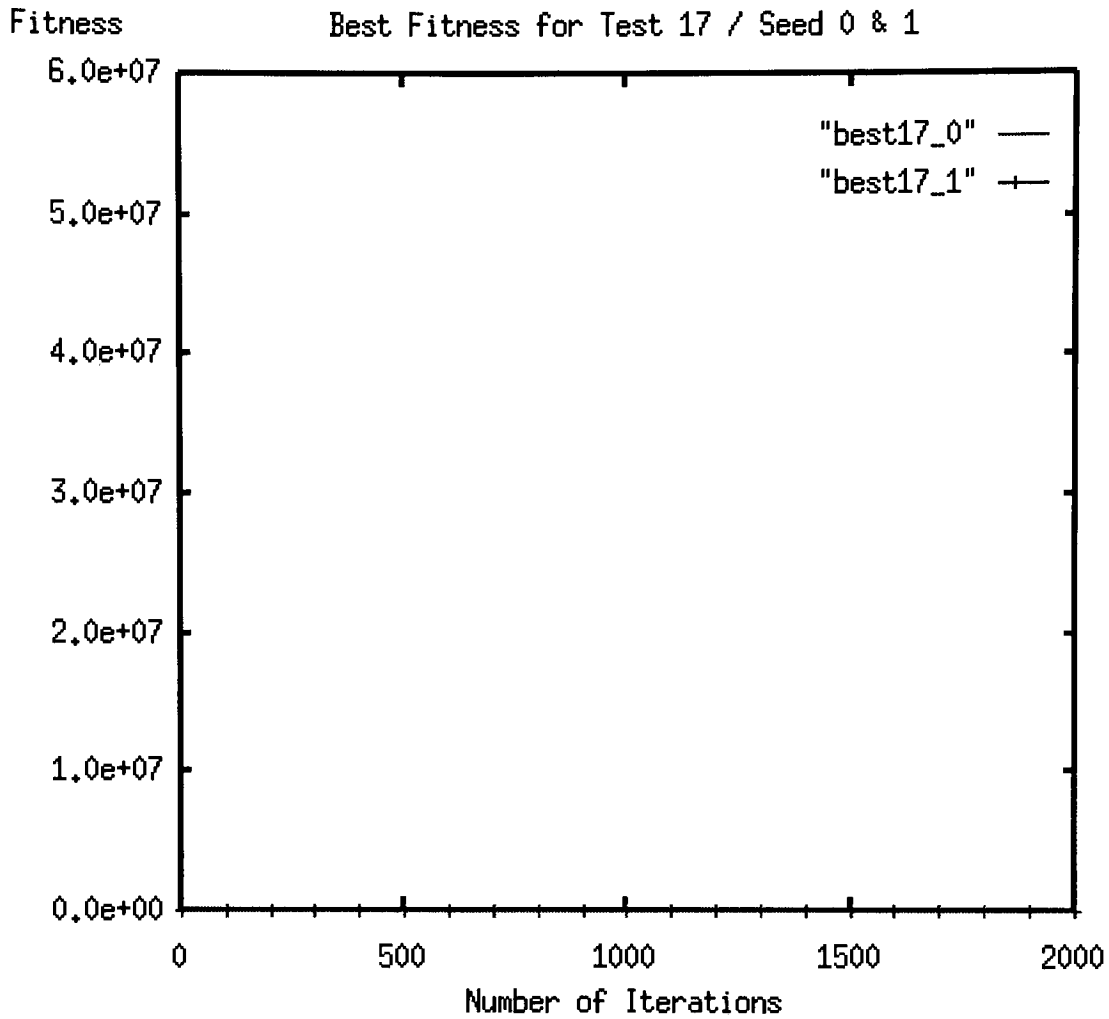
There are three levels of complexity to this map. There is food in the open, food behind water that can be found without crossing water, and food completely hidden behing water.

12.10.2 Output Data

The fitness graphs for test 17 are:



Although there were 6 pieces of food, the average fitness from the beginning is less than 5. This is due to the part of the population that came from test 9.



This graph shows that both parts of the test found all of the food before the first crossover operation. There was no need to change the solutions from test 9 to allow it to pick up heavy food. This is due to there being too many ants to solve the problem.

The solutions for this test are:

Test 17 Heavy Food and Water Crossing Problem with 100 Ants - seed 0

```
Iteration =  636, Time = Sun Apr 20 23:46:14, Fitness =    714,
     Number of Nodes = 11, Number of Turns = 714,
     Food Left = 0, Food Not Found = 0
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  MOVE INTO WATER
IF ( MOVE TO ADJACENT FOOD ) THEN
  MOVE QUASI RANDOM
  IF ( ANOTHER ANT HERE ) THEN
    PICK UP FOOD
  ELSE
    MOVE INTO WATER
ELSE
  MOVE QUASI RANDOM
```

Test 17 Heavy Food and Water Crossing Problem with 100 Ants - seed 1

```
Iteration =  1246, Time = Mon Apr 21 02:26:17, Fitness =    836,
     Number of Nodes = 9, Number of Turns = 836,
     Food Left = 0, Food Not Found = 0
MOVE QUASI RANDOM
MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE QUASI RANDOM
  ELSE
    MOVE INTO WATER
```

12.10.3 Synopsis

These are very simple solutions, and do not use any pheromone statements. There are enough ants that there is no check to make sure that all of the ants don't kill themselves in the water. Also, there are too many ants to have a deadlock problem with heavy food.

With many ants, the problem is too easy to solve. In the next test, fewer ants will be used, and hopefully the problem will still be solvable.

12.11 Water Crossing and Heavy Food Problem (Test 18)

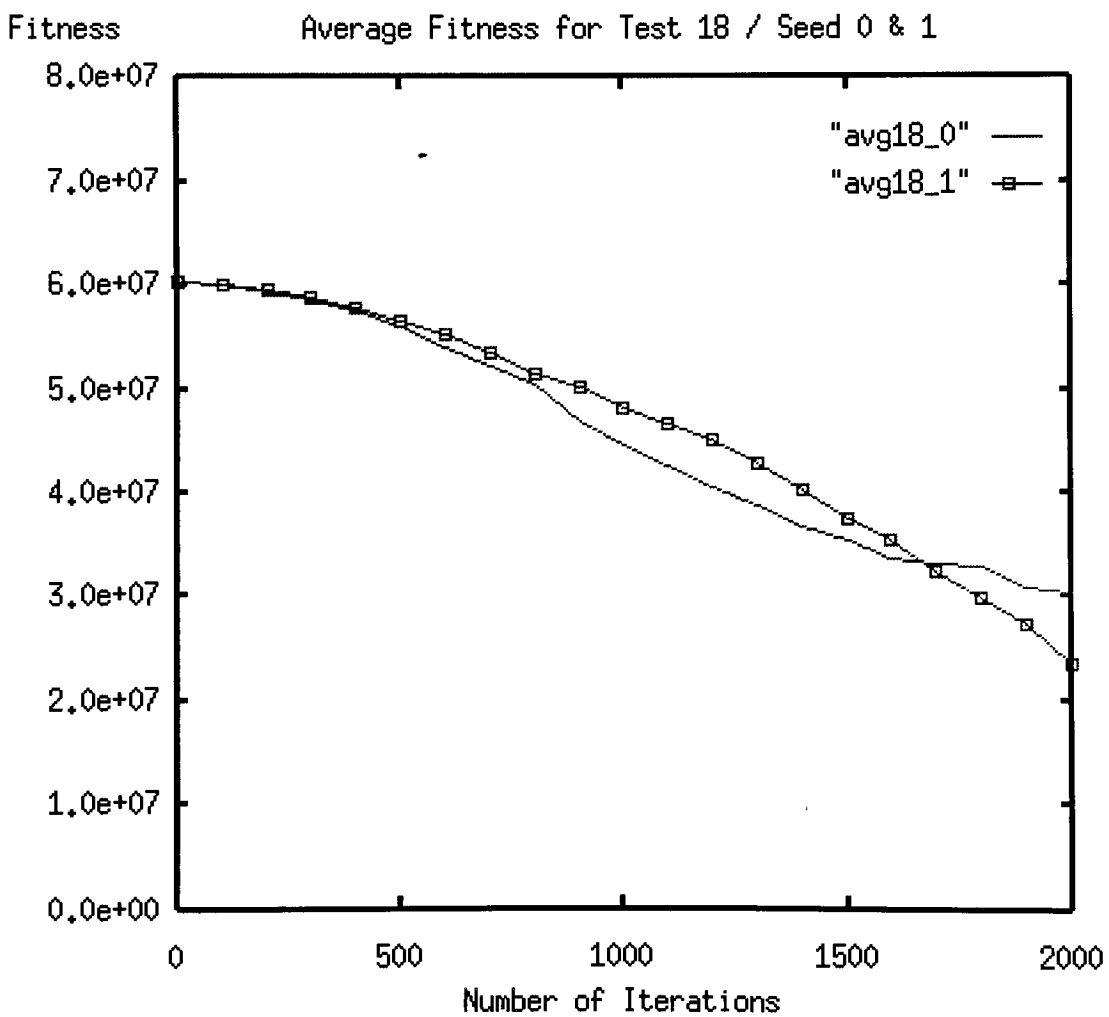
12.11.1 Input Parameters

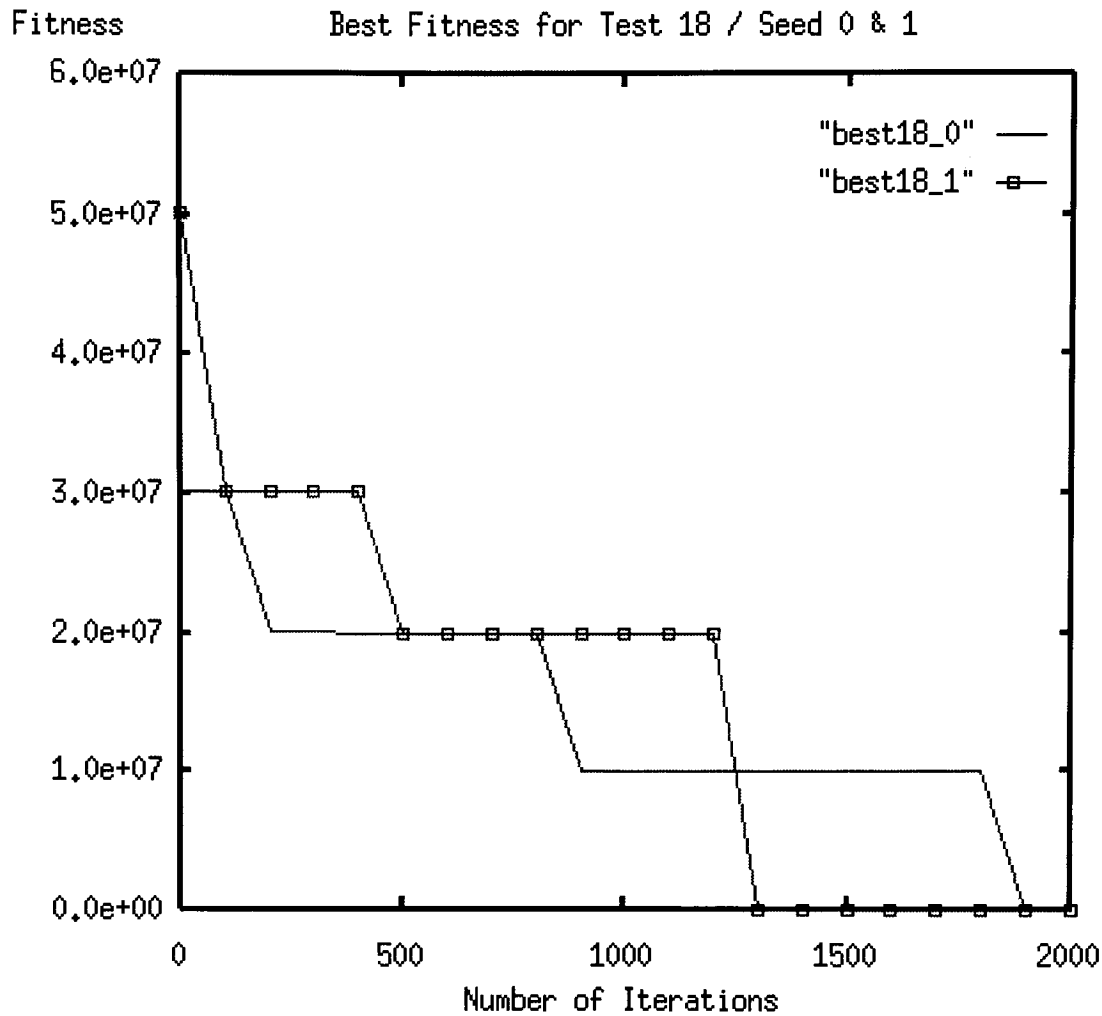
Test 18 is the same as test 17, except for 2 changes. There are not only 40 ants, not 100, and only 100 solutions from test 14 were used, not all 500. The total population was decreased to 600 to compensate for this.

Hopefully, these changes will make it more difficult on the ants. There are less ants, so there might be a deadlock problem. Also, only 1/6 of the programs are from the previous run, so all of the solutions will not look like the best solution from test 14.

12.11.2 Output Data

These are the finess graphs for test 18.





Both seeds in this test found solutions that located all of the food. In the average fitness graph, both parts of the test haven't leveled off in fitness yet.

The solutions that were found are:

Test 18 - Heavy Food and Water Crossing Problem with 40 Ants - seed 0

```
Iteration =  1858, Time = Wed Apr 23 01:37:11, Fitness =    11953,
     Number of Nodes = 87, Number of Turns = 1953,
     Food Left = 0, Food Not Found = 0
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  RELEASE PHEROMONE
MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
```

```
    RELEASE PHEROMONE
  ELSE
    MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  RELEASE PHEROMONE
  IF ( ANOTHER ANT HERE ) THEN
    MOVE INTO WATER
  ELSE
    RELEASE PHEROMONE
ELSE
  MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE INTO WATER
  ELSE
    IF ( FOOD HERE ) THEN
      IF ( CARRYING FOOD ) THEN
        IF ( MOVE TO ADJACENT FOOD ) THEN
          ESCAPE PHEROMONE
        ELSE
          MOVE TO NEST
      ELSE
        MOVE TO NEST
    ELSE
      IF ( MOVE TO AWAY PHEROMONE ) THEN
        IF ( MOVE TO DEAD ANT ) THEN
          MOVE INTO WATER
        ELSE
          MOVE QUASI RANDOM
      ELSE
        IF ( WATER AHEAD ) THEN
          IF ( MOVE TO NEST ) THEN
            IF ( FOOD HERE ) THEN
              IF ( CARRYING FOOD ) THEN
                IF ( MOVE TO ADJACENT FOOD ) THEN
                  ESCAPE PHEROMONE
                ELSE
                  MOVE TO NEST
              ELSE
                RELEASE PHEROMONE
            ELSE
              IF ( MOVE TO AWAY PHEROMONE ) THEN
                IF ( FOOD HERE ) THEN
                  MOVE INTO WATER
                ELSE
                  MOVE INTO WATER
              ELSE
                MOVE TO NEST
          ELSE
            IF ( FOOD HERE ) THEN
              IF ( MOVE TO AWAY PHEROMONE ) THEN
                PICK UP FOOD
```

```
                ELSE
                  IF ( CARRYING FOOD ) THEN
                    IF ( MOVE TO NEST ) THEN
                      PICK UP FOOD
                    ELSE
                      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
                        IF ( MOVE TO AWAY PHEROMONE ) THEN
                          IF ( MOVE TO ADJACENT FOOD ) THEN
                            ESCAPE PHEROMONE
                          ELSE
                            MOVE TO NEST
                        ELSE
                          IF ( MOVE TO AWAY PHEROMONE ) THEN
                            MOVE INTO WATER
                          ELSE
                            MOVE INTO WATER
                      ELSE
                        PICK UP FOOD
                  ELSE
                    MOVE TO NEST
                    RELEASE PHEROMONE
              ELSE
                IF ( MOVE TO DEAD ANT ) THEN
                  PICK UP FOOD
                ELSE
                  MOVE QUASI RANDOM
          ELSE
            IF ( MOVE TO ADJACENT FOOD ) THEN
              PICK UP FOOD
            ELSE
              MOVE INTO WATER
            IF ( MOVE TO ADJACENT FOOD ) THEN
              MOVE QUASI RANDOM
              IF ( ANOTHER ANT HERE ) THEN
                PICK UP FOOD
              ELSE
                RELEASE PHEROMONE
            ELSE
              MOVE QUASI RANDOM
MOVE QUASI RANDOM
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
ELSE
  IF ( MOVE TO ADJACENT FOOD ) THEN
    MOVE INTO WATER
  ELSE
    MOVE QUASI RANDOM
```

This solution is so large, therefore it was penalized in fitness for being above 50 nodes. This solution uses the method of releasing pheromones always, every turn, to spread out the ants quickly. All of the ants are moving and following the strongest pheromones away from the nest. Also, ants will not move into the water if they detect pheromones,

unless there is another ant at the same location. Therefore, if the ants get lucky, a couple of ants will get to the water, and cross it, before the pheromones reach the water. Once the pheromones, are at the water, the only way the water can be crossed is by having two ants be at the same location next to the water. Then they can both move into the water. This will limit the amount of ants that die while crossing the water. If given enough time, this solution can technically always solve the problem, as long as all of the ants don't die. In this solution, the ants must find the nest randomly when bringing the food back to the nest.

Test 18 Heavy Food and Water Crossing Problem with 40 Ants - seed 1

```
Iteration =  1918, Time = Wed Apr 23 01:31:38, Fitness =     913,
     Number of Nodes = 21, Number of Turns = 913,
     Food Left = 0, Food Not Found = 0
PICK UP FOOD
IF ( ANOTHER ANT HERE ) THEN
  MOVE INTO WATER
ELSE
  RELEASE PHEROMONE
IF ( MOVE TO ADJACENT FOOD ) THEN
  PICK UP FOOD
  IF ( ANOTHER ANT HERE ) THEN
    PICK UP FOOD
  ELSE
    RELEASE PHEROMONE
ELSE
  MOVE QUASI RANDOM
IF ( CARRYING FOOD ) THEN
  MOVE TO NEST
ELSE
  IF ( TIRED OF WAITING ) THEN
    PICK UP FOOD
  ELSE
    RELEASE PHEROMONE
PICK UP FOOD
```

This solution only crosses the water if there is another ant at the same location. This will greatly limit the number of ants that can move into the water. This solution counts on the fact that the ants will get lucky, and be at the same location at least twice, so that they can cross the water. This solution does a drunken walk back to the nest with the food. Both of these solutions do not deal with the deadlock issue. Instead, they both found ways to limit the amount of ants that die, so there is no shortage of ants to solve the problem.

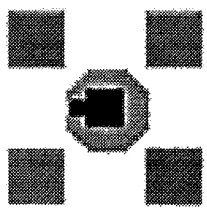
12.11.3 Synopsis

The best solutions from this test do not appear to have gained any knowledge from test 9. This test's method of crossing water was completely different. Test 9 used 100 ants to solve the problem, as did test 17 which used all 500 solution from test 9 in its initial population. This test used only 40 ants. Test 17 made good use of the information learned from test 9, while this test did not. It appears that the *RESTORE* parameters should only be used if the other parameters are also kept the same.

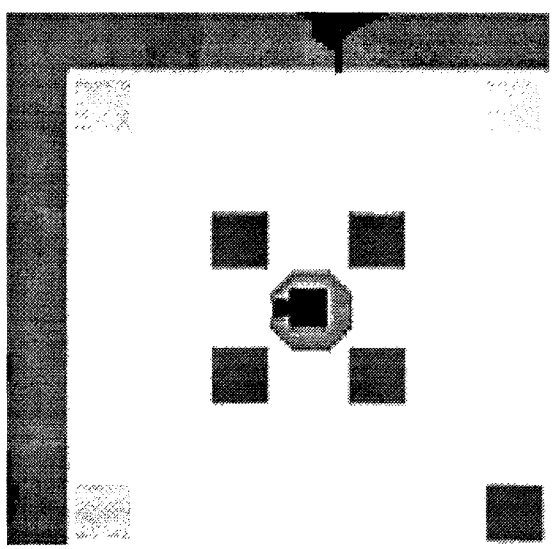
13 Unexpected Results

Many of the basic decisions that were made about the program design had a great affect on the outcome of testing. Some of these decisions were made to make the program design simpler. Others were necessary. Any decisions that caused unexpected results, along with any design issues that differed from Koza's ant problem are listed here.

The first decision was that ants would not travel along a diagonal. To move North East, the ant would move North and then East. When an ant would move to an adjacent piece of food, or an adjacent pheromone, it would only be along the main 4 axis. In Koza's example, all 8 adjacent squares were used. Since the MOVE RANDOM statement has the ant move 2 spaces in a random direction, there will be some areas where the ant cannot find food. Take the following example.

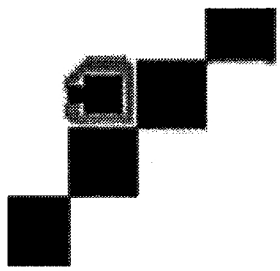


Here, the ant will never be able to be adjacent to any of the 4 pieces of food, if it moves 2 spaces at a time. It cannot find this food. But, if the nest is an odd number of spaces from the edge or an obstacle, the ant will be able to find the food when it hits the wall. Think of the map as a checker board. From the start, the ants can only move on the black squares on the even rows. Then the ant moves into a wall that is only 1 space away. The ant couldn't move 2 spaces, and now it is moving on the red squares. The ant can now find the food. In the following example, the ant will not be able to find any of the food, until it hits a wall.



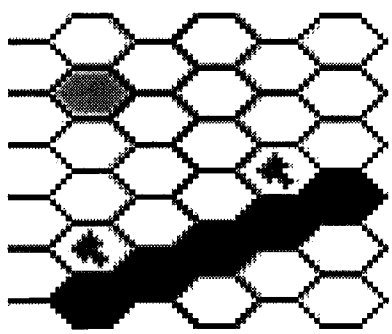
Normally, this is a bad effect. Basically, the food must be placed on spaces that the ants can reach, otherwise the problem is impossible. This has the effect of making the map smaller for all intents and purposes. Still, this feature can be used as an advantage. In the heavy food problem, if there are two pieces of food, one that the ants can find, and one that they can't, then all the ants will move toward the one piece of food. They will also release pheromones. The ants will all find the one piece of food and bring it back to the nest. Because of the pheromones, ants stuck in the pheromone cloud will not be tied to just even or odd squares. The MOVE TO ADJACENT PHEROMONE statement only moves the ant one square, not two. Thus, finding the first piece of food will set up the ants to find the second piece of food. This turns the problem into an advantage.

Still, there is the question, why can't ants move on diagonals. If the ants could move on a diagonal, then they would be able to cross a diagonal stream that was one square wide, as shown below.



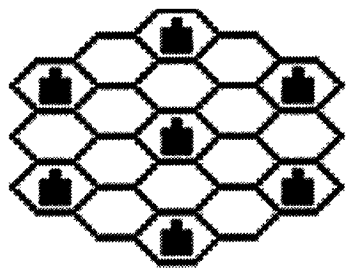
Here, the ant could move South East, and cross the water. This is not desirable.

The best solution would be to use a hex grid, instead of a square one. The ants would be able to move in one of six directions. There would also be less of a staircase effect for water and obstacles, as seen below.



In a square grid, because of the staircase effect, two ants that are next to the water are not able to move toward each other. If one ant is to the North and East of another, ant both ants are next to a stream running North-East, then the ant will have to move West before moving South to find the other ant. If the pheromones are stronger to the South, then the ant will never move.

Also, a hex grid would remove the possiblity of ants not being able to find food when they move two spaces at a time. This removes the all problems with diagonal movement and the MOVE RANDOM statement. As in the picture below, the center ant can move to any of the 6 outer positions, which allows the ant to be next to any hex space on the map.

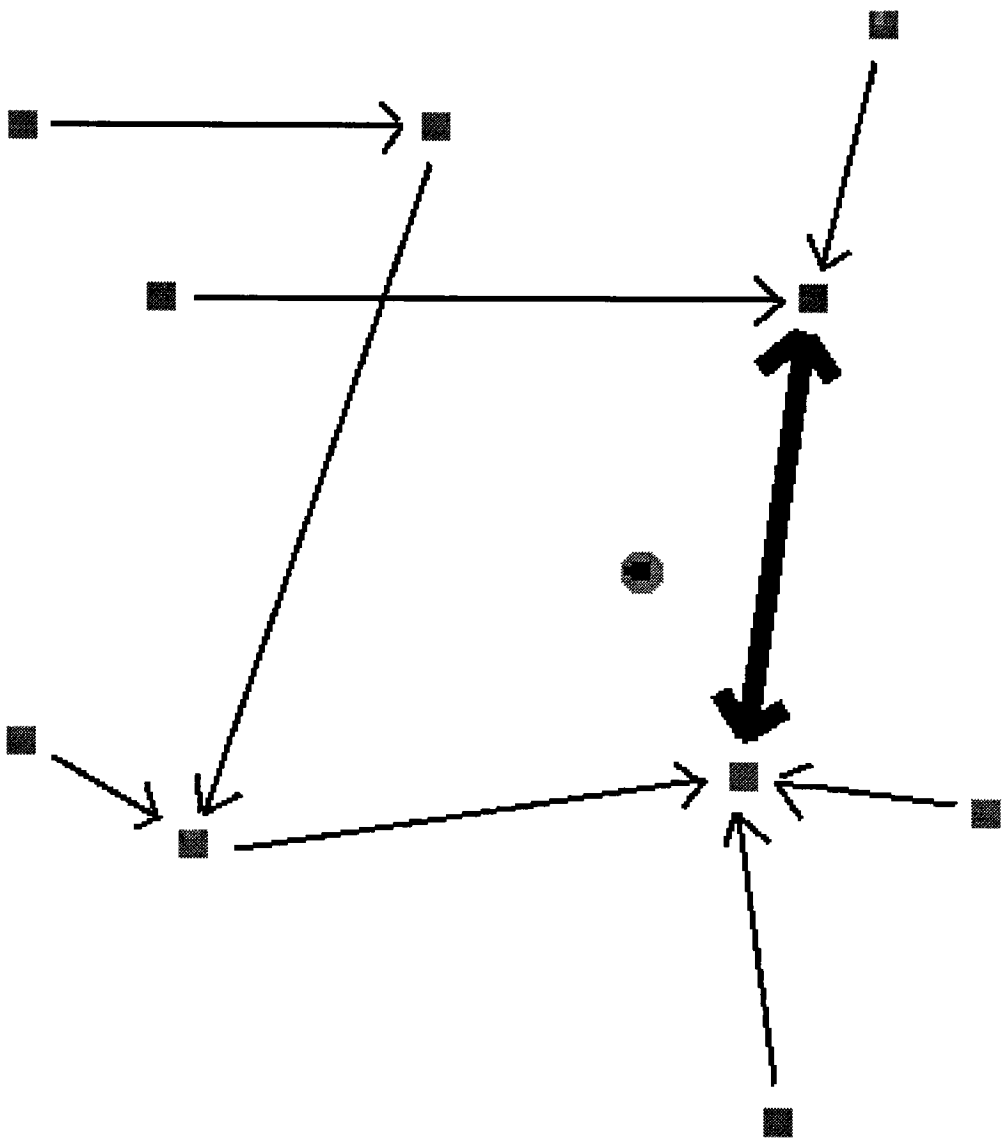


Another oddity was the ants' use of pheromones. Pheromones were initially intended to be used to mark where food is, either a cloud around a heavy piece of food, or a trail leading to a pile of food. In some tests, the ants also used pheromones to mark areas that had already been searched for food. By having every ant always release pheromones, the ants can use the pheromones to spread out faster than they normally would. Maybe a second type of pheromone smell could be added, a bad pheromone that the ants would move away from. The genetic program might find an ingenious use for this pheromone that would allow the problems to be solved faster.

The ants also used pheromones in the same way to limit the number of ants that move into the water. The ants would release pheromones every turn, and then execute the MOVE TO AWAY PHEROMONE statement. This caused the ant to create a very large pheromone cloud, and all of the ants were at the edge of the cloud. When the ants found water, if they did not detect pheromones, then they would move into the water to build a bridge. The ants just outside the edge of the cloud would build the bridge, and those just inside the cloud would not move into the water, because they detected pheromones. These ants just inside the pheromone cloud are the ones that move across the bridge. When the ants use this method, usually only 4 ants die total when building a bridge across the water.

In the heavy food problem, the ESCAPE PHEROMONE has an odd effect if there are many pieces of food on the map. If there are many ants at different pieces of food releasing pheromone clouds, then when an ESCAPE PHEROMONE instruction is executed, the ant will be moved outside the pheromone cloud. If the ant was facing in the direction of the nest, if it had tried to execute a MOVE TO NEST statement, then it will escape the pheromone cloud in the direction of the nest. Then, the ant will have a good chance of finding the pheromone cloud for a piece of food closer to the nest. The ants could all end up moving toward two pieces of food on the map.

In the picture below, there are 10 pieces of food, but the ants will all move toward the two pieces of food marked by the wide arrow. The arrows indicate which way the ant will be facing if it executes a MOVE TO NEST, and which piece of food it will move toward if it then does a MOVE TO ADJACENT pheromone. The ants do not necessarily have to find the indicated food first, but they are already heading in that direction.



14 Design Issues

One decision was that the ants would start at the nest. Originally, the nest and the starting position of the ants was different. Also, the ants could start in random positions. This appears to have no effect on the outcome of the problem, unless the ants start on one side of a stream, and the nest is on another. I decided is was more realistic to have the ants start at the nest, so that they should know how to get back to the nest.

I also decided to use the MOVE QUASI RANDOM statement in most of the tests. Testing proved that having the ant move forward half of the time allowed the problem to be solved faster than pure random movement. This seemed to mimic the behavior of real ants better than Koza's MOVE RANDOM statement.

In all of the tests, the streams had a width of two. I figured that a width of one would be too simple, while three was unnecessarily large. In the solutions for the water crossing problem, the solutions were specific for a stream of with two. The perfect water crossing solution found in test 9 seed 0 will not work at all for a stream of width 3.

As for the actual implementation of the genetic program, I decided to use C instead of LISP, because of its flexibility. One advantage of LISP is the ability to easily store the ant programs as lists. Koza did not limit his functions to binary trees, and he did not need to have ELSE clauses in his IF THEN statements. I believe that nothing is lost by limiting the programs to binary trees. Two PROC 2 statements are the equivalent of one PROC 3 statement. Also, the ELSE clause does not need to contain an actual statement. It could be a NOOP, or a PICK UP FOOD statement.

Finally, all of the tests used greedy mutation 80% of the time. After many tests, it was found that greedy mutation is almost always beneficial. It also speeds up the genetic program in allowing a solution to be found faster.

15 Conclusion

Execpt for the simple food collection problem, almost none of the tests produced expected results. In the simple food collection problem, the ants randomly searched for food, and then carried it back to the nest, leaving a pheromone trail to the pile of food. The only statement the ants didn't use was MOVE TO ADJACENT FOOD, so the ants had to land on the food to find it, instead of moving next to it. Since the ants were able to find the food without this statement, the test was a success.

In the next problem, Water Crossing, the ants used an ingenious method to cross the water. They also built a pheromone cloud around the food before heading back to the nest with food. The solution that the Genetic Program found works best when there are many ants. When an ant dies, there must be another ant nearby to notice the pheromone cloud produced by the ant before it dies. When there are many ants (around 100) then there is a good chance that another ant will be near. Although only 10 ants out of 100 might die, if the ant program is run with 30 ants, then all of the ants will die. None of them will notice the pheromones from the other ants, because they are too far apart. The solution was specifically for many ants.

In the Heavy Food Problem with deadlock, the best solutions were based on luck. Without a deadlock condition, this problem was easy to solve. With deadlock, the Genetic Program would eventually get lucky and evaluate a solution that solved the problem, because it did not run into the deadlock situation. Did the Genetic Program find a solution to the given problem? The answer is yes. Is this the solution I wanted? No. In a real life situation, you cannot have a solution that only works a small percentage of the time. The good news is that with more ants, there is a solution that works every time, but this solution does not involve any teamwork. Each ant is just acting on its own.

Finally, there is the combined Water Crossing & Heavy Food Problem. The water crossing part of this problem was not difficult. The heavy food part had the same problems as the previous problem. The Genetic Program found a perfect solution, but it relied on luck, and didn't always work. If more ants were used, say 100, and this lucky solution were evaluated again, it still might not work. But, if the same number of ants, 100, were used and the Genetic Program evolved a new solution, then the Geneitc Program would find a solution that worked all of the time.

In conclusion, the Genetic Program evolves a solution that solves the problem you give it. If the problem is too difficult, then it evolves a solution that will not work every time. Otherwise, it will find a solution that will work consistently. When the Genetic Program creates a solution for one problem, this solution may or may not be adaptable to another problem, no matter how similar it is. This was found when the *RESTORE* parameters were used.

16 Future Research

The biggest improvement that could be made to the Genetic Program would be to use a hex map, instead of squares, for the reasons discussed in the Unexpected Results Chapter. It would be a major change to the Genetic Program and the Display program code, but it would remove many problems: not being able to find food on certain squares, the staircase effect of water, and diagonal movement.

Many of the solutions found for the Water Crossing Problem were specific to maps that had a stream of width two. More research could be done with maps that had different widths. A stream of width one would be too simple, but streams of width three and more would be a more difficult problem to solve.

Other instructions could be added to the instruction set. One addition could be a second type of pheromone. The statements RELEASE PHEROMONE 2 and MOVE TO ADJACENT PHEROMONE 2 could be used. Also, the second type of pheromone could be used to repel ants, instead of attracting them. By adding the statement MOVE AWAY FROM PHEROMONE 2, the ants would have a better method of spreading themselves out. The ants could mark areas to avoid, areas that have already been searched for food.

Another element to add to the problem would be predators. If a predator could kill an ant, but multiple ants could kill a predator, then the ants would be forced to use teamwork to survive. It would be interesting to see if the Genetic Program could evolve an ant program where the ants moved around in groups of two or more for protection. More special instructions would have to be added to make this possible.

Many of the best solutions to the complex problems work only a fraction of the time. Because they work perfectly once, then to the Genetic Program they are the best solution. But, they are not perfect solutions. Some of these solutions will not find any food nine times out of ten. If the fitness testing was done on each solution more than once, and the fitnesses were averaged together, then the Genetic Program might produce more consistent solutions.

It appears that restoring a population from a previous test using the *RESTORE* parameters is not always a good idea. In Test 9, 100 ants were used in a water crossing problem. Test 17 restored 500 solutions from test 9 in a water crossing / heavy food problem, and also used 100 ants. The solutions from test 9 were able to solve the problem without change, ant the Genetic Program had a perfect solution before the first crossover. Test 18 restored 100 solutions from test 9 in a water crossing / heavy food problem, but used only 40 ants. Its best solution didn't use any part of the solutions from test 9. Simply changing the number of ants made the best solution from test 9 be a bad solution for test 18. Further research is needed to determine what other changes to the parameters and the map file will have a similar effect.

17 Bibliography

Belew, Richard K. and Booker, Lashon B. **Proceedings of the Fourth International Conference on Genetic Algorithms.** Morgan Kaufmann Publishers Inc. San Mateo, CA. 1991

Cona, John. **Developing a Genetic Programming System.** *AI Expert.* Vol 10. Num 2. Feb 95. pg 20-29.

Dietterich, Thomas G. **Learning at the Knowledge Level.** *Machine Learning.* Vol 1. Num 3. pg 287-316.

Forrest, Stephanie and Mitchell, Melanie. **What Makes a Problem Hard for a Genetic Algorithm? Some Anomalous Results and their Explanation.** *Machine Learning.* Vol 13. Num 2/3. Nov/Dec 93.

Hedberg, Sara. **Emerging Genetic Algorithms.** *AI Expert.* Vol 9. Num 9. Sept. 94. pg 25-37.

Howard, Les M. and D'Angelo, Donna J. **The GA-P: A Genetic Algorithm and Programming Hybrid.** *IEEE Expert.* Vol 10. Num 3. June 1995. pg 11-15.

Kinnear, Kenneth E. **Advances in Genetic Programming.** MIT Press. Cambridge, Mass. 1994.

Koza, John R. **Genetic Programming: On the Programming of Computers by the means of Natural Selection.** The MIT Press. 1993.

McTamaney, Louis S. **Mobile Robots - Real Time Intelligent Control.** *IEEE Expert.* Vol 2. Winter 87. pg 55-68.

Soucek, Branko. **Dynamic, Genetic, and Chaotic Programming**. John Wiley & Sons, Inc. 1992.

Tambe, Milind et. al. **Intelligent Angents for Interactive Simulation Environments**. *AI Magazine.* Vol 16. Num 1. pg 15-39.

Wong, Man Leung and Leung, Sak Kwong. **The Genetic Logic Programming System**. *IEEE Expert.* Vol 10. Num 5. Oct 95. pg 68-69.

Source Code

commands.h	defines of all instructions as numbers
scommands.h	defines of all instructions as strings
types.h	typedefs used in the code
map.h	defines pertaining to the map input file
dir.h	locations of input and output files
error.h	list of error codes
ant.c	main for the ant program - loops thru each iteration
allo.c	functions to allocate arrays
commands.c	read in the instruction file
crossover.c	perform the crossover operation
display.c	main for the display program - contains all X calls
dump.c	all output functions
error.c	process an error message
execute.c	main interp loop for display program
init.c	create the initial population
interp.c	intepret the ant program and evaluate it for fitness
map.c	read in the map file
mutate.c	perform mutation on an ant program
pop.c	perform tournament selection
save.c	save the entire population to file
tree.c	tree copy functions
linput.l	lex file for reading in an ant program
yinput.y	yacc file for reading in an ant program
make-params	create params.c
params.d	default parameter values
Makefile	make instructions
instruction1	sample input instruction file
map1	sample input map file

```
/*
** commands.h
**
** This file defines the base and sub types of the nodes in the binary
** tree.
** For the crossover operation, the base type of both nodes must be the
** same.
** The sub type further describes the node.
**
*/

/* Types */

#define LEAF_STATEMENT        100
#define NON_LEAF_STATEMENT    101

extern int numLeafs;
extern int numNonLeafs;

extern int Leafs[100];
extern int NonLeafs[100];

/* THESE MUST BE IN THE SAME ORDER AS scommands.h */

typedef enum LeafStmt {
MOVE_RANDOM = 0,
MOVE_QUASI_RANDOM,
MOVE_TO_NEST,
MOVE_TO_NEST_THROUGH_WATER,
TURN_LEFT,
TURN_RIGHT,
MOVE_FORWARD,
PICK_UP_FOOD,
DROP_FOOD,
RELEASE_PHEROMONE,
MOVE_INTO_WATER,
ESCAPE_PHEROMONE,
NOOP,
TOTAL_LEAF_STATEMENTS
} LeafStmt;

typedef enum NonLeafStmt {
PROC2 = TOTAL_LEAF_STATEMENTS,
IF_FOOD_AHEAD,
IF_FOOD_LEFT,
IF_FOOD_RIGHT,
IF_FOOD_ADJACENT,
IF_FOOD_HERE,
IF_CARRYING_FOOD,
IF_AT_NEST,
IF_MOVE_TO_ADJACENT_FOOD,
IF_MOVE_TO_AWAY_PHEROMONE,
IF_MOVE_TO_DEAD_ANT,
IF_MOVE_TO_ADJACENT_PHEROMONE,
IF_MOVE_TO_NEST,
IF_MOVE_TO_NEST_THROUGH_WATER,
IF_WATER_AHEAD,
IF_OBSTACLE_AHEAD,
IF_ANOTHER_ANT_HERE,
```

```
IF_MOVE_INTO_WATER,
IF_CANT_LIFT_FOOD,
IF_TIRED_OF_WAITING,
TOTAL_NON_LEAF_STATEMENTS
} NonLeafStmt;

typedef enum OtherStmt {
ELSE = TOTAL_NON_LEAF_STATEMENTS,
TAB,
START,
END
} OtherStmt;
```

```
/*
**
** scommands.h
**
** THESE MUST BE IN THE SAME ORDER AS commands.h
**
*/

char *CommandText[] = {

"MOVE RANDOM",
"MOVE QUASI RANDOM",
"MOVE TO NEST",
"MOVE TO NEST THROUGH WATER",
"TURN LEFT",
"TURN RIGHT",
"MOVE FORWARD",
"PICK UP FOOD",
"DROP FOOD",
"RELEASE PHEROMONE",
"MOVE INTO WATER",
"ESCAPE PHEROMONE",
"DO NOTHING",

"PROC2",
"IF ( FOOD AHEAD ) THEN",
"IF ( FOOD LEFT ) THEN",
"IF ( FOOD RIGHT ) THEN",
"IF ( FOOD ADJACENT ) THEN",
"IF ( FOOD HERE ) THEN",
"IF ( CARRYING FOOD ) THEN",
"IF ( AT NEST ) THEN",
"IF ( MOVE TO ADJACENT FOOD ) THEN",
"IF ( MOVE TO AWAY PHEROMONE ) THEN",
"IF ( MOVE TO DEAD ANT ) THEN",
"IF ( MOVE TO ADJACENT PHEROMONE ) THEN",
"IF ( MOVE TO NEST ) THEN",
"IF ( MOVE TO NEST THROUGH WATER ) THEN",
"IF ( WATER AHEAD ) THEN",
"IF ( OBSTACLE AHEAD ) THEN",
"IF ( ANOTHER ANT HERE ) THEN",
"IF ( MOVE INTO WATER ) THEN",
"IF ( CANT LIFT FOOD ) THEN",
"IF ( TIRED OF WAITING ) THEN",

"ELSE",
"    ",
"{",
"}"

};
```

```
/*
** types.h
**
** The files defines the basic tree structure to store the genetic program
**
*/

#include "map.h"

typedef struct {
      int   x;
      int   y;
} POINTTYPE;

typedef struct tnode*TreePtr;

typedef struct tnode {
      int         type;
      TreePtr     left;
      TreePtr     right;
} TreeType;

typedef struct {
      TreePtrtree;
      int   fitness;
      int   foodLeft;
      int   foodNotFound;
      int   numNodes;
      int   numTurns;
} ProgramType ;

typedef ProgramType *PopType;

typedef struct {
      int         foodCarried;
      DIRS        dir;
      int         alive;
      POINTTYPE   loc;
} AntType;

typedef AntType *AntsType;

typedef struct MapType {
      OBJECTS     object;
      int         food;
      int         pheromone;
} MapType;

typedef struct FoodIDType {
      int         weight;
      int         numAntsLifting;
      int         foundThisTurn;
      int         moved;
      int         leader;
      int         found;
} FoodIDType;

#define NO_FOOD-1
```

```
#define NULL_TREE(TreePtr) 0
```

```
/*
**
** map.h
**
** header file for map data
**
*/

typedef enum DIRS {
      NORTH = 0,
      EAST,
      SOUTH,
      WEST
} DIRS;

typedef enum OBJECTS {
      NOTHING= 0,
      DEAD_ANT,
      WALL,
      WATER,
      NEST
} OBJECTS;

typedef enum OBJECT_TYPE {
      OBJECT = 1,
      FOOD,
      PHEROMONE
} OBJECT_TYPE;

#define STARTX    NESTX
#define STARTY    NESTY

#define MAPX    40
#define MAPY    40

#define NUM_VALS5

#define PIXELS13

#define MAX_ANTS100
#define MAX_FOOD50
```

```
/*
**
** dir.h
**
**
*/

#define PIXMAP_FILE           "ant/src/pixmap.xbm"
#define PARAMS_FILE           "ant/data/params"
#define FIRST_FILE            "ant/data/first"
#define MAP_FILE              "ant/data/map"
#define INSTRUCTION_FILE      "ant/data/instruction"
#define RESULTS_DIR           "ant/results/"
#define TEST_FILE             "ant/results/test"
```

```
/*
**
** error.h
**
*/

typedef enum ErrorCode {
      E_DONE = 0,
      E_NO_PARAM_FILE_SPECIFIED,
      E_OPEN_PARAM_FILE,
      E_ALLOC,
      E_MAP,
      E_TWO_NEST,
      E_ILLEGAL_INSTRUCTION,
      E_OPEN_INSTRUCTION_FILE,
      E_INSTRUCTION_SETTING,
      E_OPEN_RESULT,
      E_NULL_TREE,
      E_NOT_NEW_VERSION,
      E_FIRST_FILE,
      E_DIRECTION,
      E_MAP_FILE,
      E_WRITE_SAVE_FILE,
      E_GET_SAVE_FILE,
      E_SAVE_FILE,
      E_SAME_VERSION,
      E_LEX,
      E_BAD_INPUT_SPACING
} ErrorCode;

#define ERROR( _xxx1 ) \
    error( __FILE__, __LINE__, _xxx1 );
```

```
/*
**
** ant.c
**
** This is the main for the ant genetic program.
**
**
*/

#include <sys/time.h>
#include "error.h"
#include "params.h"
#include "types.h"

#define NUM_AVG 64

extern PopType    population;

int   itteration;
int   bestIndividual;

void main(argc, argv)
int argc;
char *argv[];
{
    int i;
    double avg;
    double oldAvg;
    double tmpAvg;
    int lastBest, lastBestNodes;
    int prevPopSize;
    char *stime;
    time_t tim;

    params( argc, argv );

    instr();

    if ( !readMap() )
      ERROR( E_MAP );

    if ( VERSION == RESTORE_VERSION ) {
      ERROR( E_SAME_VERSION );
    }

    paramDump();

    srand48( POP_SEED );/* seed random number generator */

    setUpVars();

    if ( RESTORE_AMOUNT ) {
      popRestore( &prevPopSize );
      initialPopulation( prevPopSize );
    } else {
      prevPopSize = ( first( &population[0] )) ? 1 : 0;
      initialPopulation( prevPopSize );
    }
```

```
    findBest( &bestIndividual );

    itteration = 0;

    lastBest = population[ bestIndividual ].fitness;
    lastBestNodes = population[ bestIndividual ].numNodes;

    fileStringTreeDump( population[ bestIndividual ] );

    do {
      if ( ( itteration % SHOW_TIME ) == 0 ) {
          for ( i = 0, avg = 0, tmpAvg = 0; i < POPULATION_SIZE; i++ )
          {
            oldAvg = avg;
            avg += population[ i ].fitness;
            if ( avg < oldAvg )
            {
                printf("Avg variable overflow\n");
            }
            if (( ( i % NUM_AVG ) == 0 ) && ( i != 0 ))
            {
                tmpAvg = (( avg / NUM_AVG ) + ( tmpAvg * (( i / NUM_AVG )
- 1 ))) /
                                        ( i / NUM_AVG );
                avg = 0;
            }
          }
          avg = (( avg / ( POPULATION_SIZE % NUM_AVG )) +
                  ((( POPULATION_SIZE / ( POPULATION_SIZE %
                  NUM_AVG )) - 1 ) * tmpAvg )) / ( POPULATION_SIZE /
                  ( POPULATION_SIZE % NUM_AVG ));

          tim = time(0);
          stime = ctime(&tim);
          stime[19] = '\0';
         printf("Iteration = %6d, Time = %s,\n\tAverage Fitness = %9.0f,
Best Fitness  = %9d\n",
                  itteration, stime, avg, population[ bestIndividual
].fitness );

      }
      if ( ( itteration % DUMP ) == 0 ) {
          popSave();
      }
      itteration++;
      makeChildren( &bestIndividual, lastBest);
      if (( population[ bestIndividual ].fitness < lastBest ) ||
          (( population[ bestIndividual ].fitness == lastBest ) &&
           ( population[ bestIndividual ].numNodes < lastBestNodes ))) {
          fileStringTreeDump( population[ bestIndividual ] );
          lastBest = population[ bestIndividual ].fitness;
          lastBestNodes = population[ bestIndividual ].numNodes;
      }
   } while (( population[ bestIndividual ].fitness > GOAL_FITNESS ) && (
itteration <= MAX_ITERATIONS ));
}
```

```
/*
**
** allo.c
**
** These are the allocate functions to use params
**
*/

#include "types.h"
#include "error.h"

int alloVector( Max, size )
int Max;
int size;
{
    int *result;

    result = (int *) malloc( Max * size );
    if ( !result ) {
      printf("allocation failure in alloVector()\n");
      ERROR( E_ALLOC );
    } else
      return (int) result;
}

dalloVector( vec )
int vec;
{
   free( vec );
}

int alloMem( size )
unsigned size;
{
    int *result;

    result = (int *) malloc( size );
    if ( !result ) {
      printf("allocation failure in alloMem()\n");
      ERROR( E_ALLOC );
    } else
      return (int) result;
}

dallocTree( tree )
TreePtr tree;
{
    if ( tree )
        dallocTreeRecur( tree );
    else
      printf("Null Tree dalloced.\n");
}

dallocTreeRecur( tree )
TreePtr tree;
{
    if ( tree->right )
        dallocTreeRecur( tree->right );
    if ( tree->left )
```

```
        dallocTreeRecur( tree->left );

    free( tree );
}
```

```
/*
** commands.c
**
** get the instruction set
**
*/

#include <stdio.h>
#include "params.h"
#include "types.h"
#include "commands.h"
#include "scommands.h"
#include "dir.h"
#include "error.h"

int numLeafs  = 0;
int numNonLeafs  = 0;

int Leafs[100];
int NonLeafs[100];

extern FILE *fpi;

int isLeaf( type )
int type;
{
    return( (( type >= 0 ) && ( type < TOTAL_LEAF_STATEMENTS )) ? 1 : 0 );
}

int isNonLeaf( type )
int type;
{
    return( (( type >= TOTAL_LEAF_STATEMENTS ) &&
            ( type <  TOTAL_NON_LEAF_STATEMENTS )) ? 1 : 0 );
}

int getLeafStmt()
{
    return( Leafs[ lrand48() % numLeafs ] );
}

int getNonLeafStmt()
{
    return( NonLeafs[ lrand48() % numNonLeafs ] );
}

void AddCommand( commandStr )
char *commandStr;
{
    int i;

    for ( i = 0; i < TOTAL_NON_LEAF_STATEMENTS; i++ )
      if ( strcmp( commandStr, CommandText[i] ) == 0 ) {
          if ( isLeaf( i ) )
            Leafs[ numLeafs++ ] = i;
          else if ( isNonLeaf( i ) )
            NonLeafs[ numNonLeafs++ ] = i;
          else {
```

```
            printf( "Unknown command in instruction file: %s\n", command-
Str);
            ERROR( E_ILLEGAL_INSTRUCTION );
          }
          return;
      }

    printf( "Unknown command in instruction file: %s\n", commandStr);
    ERROR( E_ILLEGAL_INSTRUCTION );
}

instr()
{
    char filename[100];
    char commandStr[100];
    char ch;
    int  instrOn;
    int  lineNo = 1;

    sprintf( filename, "%s%d", INSTRUCTION_FILE, INSTRUCTION_VERSION );

    if (( fpi = fopen( filename, "r")) == 0 ) {
        printf("error opening file %s for read.\n", filename);
      ERROR( E_OPEN_INSTRUCTION_FILE );
    }

    do {
      ch = fgetc( fpi );

      if ( ch == EOF )
          continue;
      if ( ch == '1' )
          instrOn = 1;
      else if ( ch == '0' )
          instrOn = 0;
      else if ( ch == '\n' ) {
          lineNo++;
          continue;
      }
      else {
          printf("Illegal instruction setting in file %s at line %d\n",
filename, lineNo );
          ERROR( E_INSTRUCTION_SETTING );
      }

      ch = fgetc( fpi );

      fgets( commandStr, 100, fpi );

      commandStr[ strlen( commandStr ) - 1 ] = NULL;

      if ( instrOn )
          AddCommand( commandStr );

      lineNo++;

    } while ( ch != EOF );
}
```

```
/*
**
** crossover.c
**
** The crossover()  function will perform the crossover for the genetic
program.
** The individual programs are in the form of a binary tree.  Only nodes
with
**    the same baseType can be switched.
**
*/

#include <math.h>
#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

externint   itteration;

void crossover(individual1, individual2, child1, child2)
ProgramType individual1;
ProgramType individual2;
ProgramType *child1;
ProgramType *child2;
{
    TreePtr tree1, tree2, treeltemp, tree2temp, temp1, temp2,
treelold,tree2old;
    int side1, side2;
    int rnd;
    int  crossoverOdds;

    copyTree( individual1.tree, &tree1 );
    copyTree( individual2.tree, &tree2 );
    dallocTree( (*child1).tree );
    dallocTree( (*child2).tree );

    treeltemp = tree1;
    side1 = -1;
   crossoverOdds = (int) log( (double) individual1.numNodes ) / log( 2 )
+ 1;
    if ( crossoverOdds < 3 )
      crossoverOdds = 3;
   while ( ( lrand48() % crossoverOdds ) && ( treeltemp->left != NULL_TREE
) ) {
      rnd = lrand48() % 2;
      switch ( rnd ) {
          case 0 :
            if ( treeltemp->left != NULL_TREE ) {
                side1 = 1;
                treelold = treeltemp;
                treeltemp = treeltemp->left;
            }
            break;
          case 1 :
            if ( treeltemp->right != NULL_TREE ) {
                side1 = 0;
                treelold = treeltemp;
                treeltemp = treeltemp->right;
```

```
            }
            break;
          default :
            printf("bad case in switch\n");
      }
      rnd = lrand48() % 2;
    }

    tree2temp = tree2;
    side2 = -1;
   crossoverOdds = (int) log( (double) individual2.numNodes ) / log( 2 )
+ 1;
    if ( crossoverOdds < 3 )
      crossoverOdds = 3;
    rnd = lrand48() % 2;
   while ( ( lrand48() % crossoverOdds ) && ( tree2temp->left != NULL_TREE
) ) {
      rnd = lrand48() % 2;
      switch ( rnd ) {
          case 0 :
            if ( tree2temp->left != NULL_TREE ) {
                side2 = 1;
                tree2old = tree2temp;
                tree2temp = tree2temp->left;
            }
            break;
          case 1 :
            if ( tree2temp->right != NULL_TREE ) {
                side2 = 0;
                tree2old = tree2temp;
                tree2temp = tree2temp->right;
            }
            break;
          default :
            printf("bad case in switch\n");
      }
    }

    switch ( side1 ) {
      case -1 :
          temp2 = tree1;
          break;
      case 0 :
          temp2 = tree1old->right;
          break;
      case 1 :
          temp2 = tree1old->left;
          break;
      default : printf("crossover err - side 1 = %d\n", side1);
    }
    switch ( side2 ) {
      case -1 :
          temp1 = tree2;
          break;
      case 0 :
          temp1 = tree2old->right;
          break;
      case 1 :
          temp1 = tree2old->left;
```

```
            break;
      default : printf("crossover err - side 2 = %d\n", side1);
    }

    switch ( side1 ) {
      case -1 :
            tree1 = temp1;
            break;
      case 0 :
            tree1old->right = temp1;
            break;
      case 1 :
            tree1old->left = temp1;
            break;
      default : printf("crossover err - side 1 = %d\n", side1);
    }

    switch ( side2 ) {
      case -1 :
            tree2 = temp2;
            break;
      case 0 :
            tree2old->right = temp2;
            break;
      case 1 :
            tree2old->left = temp2;
            break;
      default : printf("crossover err - side 2 = %d\n", side2);
    }

    (*child1).tree = tree1;
    (*child2).tree = tree2;
}



/*
** debug.c
**
** print routine for debugging.
**
*/

#include <stdio.h>
#include <stdarg.h>
#include "params.h"
#include "error.h"

void dprintf(const char *fmt,...)
{
    va_list ap;
    char *p, *sval;
    int ival;

    if ( DEBUG ) {
      va_start( ap, fmt );
      for ( p = (char *)fmt; *p; p++ ) {
            if (*p != '%' ) {
              putchar(*p);
```

```
            continue;
          }
          switch (*++p) {
            case 'd' :
                ival = va_arg(ap, int);
                printf("%d", ival);
                break;
            case 'c' :
                ival = va_arg(ap, int);
                printf("%c", ival);
                break;
            case 's' :
                for (sval = va_arg(ap, char *); *sval; sval++)
                  putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
          }
        }
        va_end( ap );
    }
}
```

```
/*
** display.c
**
** This is the display process.
*/

#include <Xm/Xm.h>
#include <Xm/BulletinB.h>
#include <Xm/DrawingA.h>
#include <Xm/PushB.h>
#include <Xm/ToggleB.h>
#include <Xm/Form.h>
#include <Xm/Frame.h>
#include <Xm/Label.h>
#include <Xm/TextF.h>
#include <Xm/Text.h>
#include <Xm/Scale.h>
#include <X11/cursorfont.h>
#include <sys/types.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <math.h>

#include "dir.h"
#include "types.h"
#include "commands.h"
#include "params.h"
#include "error.h"

#define index(j,i) ((j)*(MAPY)+i)

#define loc(j,i) (j)*PIXELS, (i)*PIXELS, PIXELS, PIXELS
#define big_loc(j,i) (j)*PIXELS-2, (i)*PIXELS-2, PIXELS+4, PIXELS+4
#define small_loc(j,i) (j)*PIXELS+1, (i)*PIXELS+1, PIXELS-2, PIXELS-2

extern int NUM_FOOD;
extern int food_in_nest;
extern int foodNotFound;
extern int loops;
extern FoodIDType FOODID[ MAX_FOOD ];
extern FoodIDType foodId[ MAX_FOOD ];


extern FILE *fpi;
extern TreePtr firstProg;
extern MapType map[ MAPX ][ MAPY ];
extern MapType MAP[ MAPX ][ MAPY ];
extern int turns;
extern AntsType ants;
extern int NESTX;
extern int NESTY;

extern int lineNo;

int numOfNodes;
```

```
int legalFile = 0;
int legalMap = 0;
int didRun = 0;
int running = 0;

int itteration;
int antSpeed;

int lastFoodInNest;

char fileName[20];

ProgramType person;

int status;
int WINWIDTH, WINHEIGHT;

char *labels[ NUM_VALS ] =
      { "Fitness:", "Num Nodes:", "Num Turns:", "Food Left:", "Food Not
Found:" };
char vals[ NUM_VALS ][10] =
      { "", "", "", "", "" };

XtAppContextapp;

void        file_callback(),
            execute_callback(),
            maxTurns_callback(),
            numAnts_callback(),
            speed_callback(),
            mapNum_callback(),
            pheromoneDrop_callback(),
            pheromoneSpread_callback(),
            drawArea_callback(),
            exit_callback();

Widget      toplevel,
            bboard,
            exit_button,
            execute_button,
            pheromoneDrop_scale,
            pheromoneSpread_scale,
            maxTurns_scale,
            numAnts_scale,
            speed_scale,
            mapNum_scale,
            fileName_label,
            fileName_text,
            labels_text[ NUM_VALS ],
            vals_text[ NUM_VALS ],
            draw_frame,
            draw_area;

Display     *display;
Window      window;
Screen      screen;
Drawabledrawable;
Pixmap      pixmap;
```

```
GC          gc;

Arg args[2] ;

setVal( val, num )
int val;
int num;
{
    char tmpStr[30];
    XmString tmpXmStr;

    sprintf( tmpStr, "%d", num );
    tmpXmStr = XmStringCreate( tmpStr, XmSTRING_DEFAULT_CHARSET );
    XtSetArg ( args[0], XmNlabelString, tmpXmStr ) ;
    XtSetValues ( vals_text[val], args, 1 ) ;
    XmStringFree( tmpXmStr );
}

void set_color(widget, color)
Widget widget;
char *color;
{
    Display *dpy = XtDisplay (widget);
    Colormap cmap = DefaultColormapOfScreen (XtScreen (widget));
    XColor col, unused;

    if (!XAllocNamedColor (dpy, cmap, color, &col, &unused)) {
        char buf[32];
        sprintf (buf, "Can't alloc %s", color);
        XtWarning (buf);
        return;
    }
    XSetForeground (dpy, gc, col.pixel);
}

void drawAnt( j, i, dir )
int j;
int i;
DIRS dir;
{
    XFillRectangle( XtDisplay(draw_area), pixmap, gc, (j)*PIXELS+3,
(i)*PIXELS+3, 7, 7 );
    switch ( dir ) {
      case NORTH :
         XFillRectangle( XtDisplay(draw_area), pixmap, gc, (j)*PIXELS+5,
(i)*PIXELS, 3, 3 );
          break;
      case EAST :
        XFillRectangle( XtDisplay(draw_area), pixmap, gc, (j)*PIXELS+10,
(i)*PIXELS+5, 3, 3 );
          break;
      case SOUTH :
         XFillRectangle( XtDisplay(draw_area), pixmap, gc, (j)*PIXELS+5,
(i)*PIXELS+10, 3, 3 );
          break;
      case WEST :
          XFillRectangle( XtDisplay(draw_area), pixmap, gc, (j)*PIXELS,
(i)*PIXELS+5, 3, 3 );
          break;
```

```
    }
}

void initialMapDraw()
{
    int i, j;

    set_color (draw_area, "white");
   XFillRectangle( XtDisplay(draw_area), pixmap, gc, 0, 0, MAPX*PIXELS,
MAPY*PIXELS );

    for ( i = 0; i < MAPY; i++ ) {
      for ( j = 0; j < MAPX; j++ ) {
          switch ( MAP[j][i].object ) {
            case NOTHING :
                break;
            case WALL :
                set_color (draw_area, "red");
              XFillRectangle( XtDisplay(draw_area), pixmap, gc, loc(j,i)
);
                break;
            case WATER :
                set_color (draw_area, "blue");
              XFillRectangle( XtDisplay(draw_area), pixmap, gc, loc(j,i)
);
                break;
            case NEST :
                set_color (draw_area, "goldenrod2");
                if (( NESTX != 0 ) && ( NESTY != 0 ))
                  XFillArc( XtDisplay(draw_area), pixmap, gc,
big_loc(j,i), 0, 64*360 );
                else
                  XFillArc( XtDisplay(draw_area), pixmap, gc, loc(j,i),
0, 64*360 );
          }
          if ( MAP[j][i].food != NO_FOOD ) {
                set_color (draw_area, "green");
                XFillRectangle( XtDisplay(draw_area), pixmap, gc,
small_loc(j,i) );
          }
      }
    }


    set_color (draw_area, "brown");
    drawAnt( STARTX, STARTY, lrand48() % 4 );

    XCopyArea (XtDisplay(draw_area), pixmap, XtWindow(draw_area),
                gc, 0, 0, MAPX*PIXELS, MAPY*PIXELS, 0, 0);
}

int getHeader()
{
      char header[200];
      char *tmp;
      int tempNum;
      char ch1, ch2;
      int i, j;
```

```
      ch1 = getc( fpi );
      ch2 = getc( fpi );

      ungetc( ch2, fpi );
      ungetc( ch1, fpi );

      if (( ch1 != 'I' ) || ( ch2 != 't' )) {
            for ( i = 0; i < NUM_VALS; i++ )
                  vals[i][0] = NULL;
            return( 0 );
      }

      if ( fgets( header, 200, fpi ) == (char *) EOF )
            return( -1 );

      tmp = strchr( header, '=' );
      tmp++;
      tmp = strchr( header, '=' );
      tmp++;
      sscanf( tmp, "%d", &person.fitness );

      if ( fgets( header, 200, fpi ) == (char *) EOF )
            return( -1 );

      tmp = strchr( header, '=' );
      tmp++;
      sscanf( tmp, "%d", &person.numNodes );
      tmp = strchr( header, '=' );
      tmp++;
      sscanf( tmp, "%d", &person.numTurns );

      if ( fgets( header, 200, fpi ) == (char *) EOF )
            return( -1 );

      tmp = strchr( header, '=' );
      tmp++;
      sscanf( tmp, "%d", &person.foodLeft );
      tmp = strchr( header, '=' );
      tmp++;
      sscanf( tmp, "%d", &person.foodNotFound );

      numOfNodes = atoi( vals[1] );
}

void changeState()
{
    XmString tempXmStr;

    if ( running )
      tempXmStr = XmStringCreate( "Halt", XmSTRING_DEFAULT_CHARSET );
    else if ( MAX_TURNS <= (( loops > turns/NUM_ANTS ) ? loops : turns/
NUM_ANTS ))
      tempXmStr = XmStringCreate( "Complete", XmSTRING_DEFAULT_CHARSET );
    else if ( legalFile )
      tempXmStr = XmStringCreate( "Execute", XmSTRING_DEFAULT_CHARSET );
    else
      tempXmStr = XmStringCreate( "Need File Name",
XmSTRING_DEFAULT_CHARSET );
```

```
    XtVaSetValues( execute_button, XmNlabelString, tempXmStr, NULL );
    XtVaSetValues( execute_button, XmNwidth, 100, XmNheight, 40, NULL );
    XmStringFree( tempXmStr );
}

void setAutoFoodDrop( tree )
TreePtr tree;
{
    if (( tree == NULL_TREE ) || ( !AUTO_FOOD_DROP ))
      return;

    if ( tree->type == DROP_FOOD )
      AUTO_FOOD_DROP = 0;

    setAutoFoodDrop( tree->left );
    setAutoFoodDrop( tree->right );
}

int savedFile( testFile )
char *testFile;
{
      char fileName[100];
      int fitnessKnown;

      if ((( strncmp( testFile, "test", 4 ) == 0 ) && ( strlen( testFile
) == 6 ))
                  || ( testFile == NULL ) || ( testFile[0] == NULL ))
            return(0);

      sprintf( fileName, "%s%s", RESULTS_DIR, testFile );

      if (( fpi = fopen( fileName, "r" )) == 0 ) {
            return(0);
      }

      if (( fitnessKnown = getHeader()) == -1 ) {
            printf("Bad header format\n");
            return(0);
      }

      lineNo = 2;

      if (yyparse()) {
            printf("Bad program format\n");
            return(0);
      }

      person.tree = firstProg;

      if ( ! fitnessKnown )
            interp( &person );

      setVal( 0, person.fitness );
      setVal( 1, person.numNodes );
      setVal( 2, person.numTurns );
      setVal( 3, person.foodLeft );
      setVal( 4, person.foodNotFound );

      legalFile = 1;
```

```
      didRun = 0;
      running = 0;
      turns = 0;
      loops = 0;

      changeState( person.tree );

      AUTO_FOOD_DROP = 1;
      setAutoFoodDrop( person.tree );

      return(1);
}

void makeWidgets(argc, argv)
int argc;
char *argv[];
{
    int i;

    WINWIDTH = MAPX*PIXELS+230;
    WINHEIGHT = MAPY*PIXELS+30;
    if ( WINHEIGHT < 700 )
      WINHEIGHT = 700;

    toplevel = XtVaAppInitialize(&app, "Hello", NULL, 0,
            &argc, argv, NULL,
            XmNtitle, "ANT DISPLAY",
            XtVaTypedArg, XmNiconPixmap, XmRString, PIXMAP_FILE, strlen(
PIXMAP_FILE ),
            XmNminWidth, WINWIDTH,
            XmNmaxWidth, WINWIDTH,
            XmNwidth,    WINWIDTH,
            XmNminHeight, WINHEIGHT,
            XmNmaxHeight, WINHEIGHT,
            XmNheight,    WINHEIGHT,
             NULL);

    bboard = XtVaCreateManagedWidget("bboard",
            xmBulletinBoardWidgetClass, toplevel,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);

    exit_button = XtVaCreateManagedWidget("Exit",
            xmPushButtonWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XmNx, 60,
            XmNy, WINHEIGHT-60,
            XmNwidth, 80,
            XmNheight, 50,
            XtVaTypedArg, XmNbackground, XmRString, "gray60", 7,
            NULL);
   XtAddCallback(exit_button, XmNactivateCallback, exit_callback, NULL);

    execute_button = XtVaCreateManagedWidget("Execute",
            xmPushButtonWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XmNx, 50,
            XmNy, WINHEIGHT-110,
            XmNwidth, 100,
```

```
            XmNheight, 40,
            XtVaTypedArg, XmNbackground, XmRString, "gray60", 7,
            NULL);
   XtAddCallback(execute_button, XmNactivateCallback, execute_callback,
NULL);

    mapNum_scale = XtVaCreateManagedWidget("mapNumScale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, "    Map Version",
16,
            XmNmaximum, 40,
            XmNminimum, 1,
            XmNscaleMultiple, 1,
            XmNvalue, MAP_VERSION,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 50,
            XmNy, WINHEIGHT - 510,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
  XtAddCallback(mapNum_scale, XmNvalueChangedCallback, mapNum_callback,
NULL);

    pheromoneSpread_scale = XtVaCreateManagedWidget("pheromoneSpread-
Scale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, "Pheromone Spread
Percent", 25,
            XmNmaximum, 100,
            XmNminimum, 0,
            XmNscaleMultiple, 10,
            XmNvalue, PHEROMONE_SPREAD,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 20,
            XmNy, WINHEIGHT - 445,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
    XtAddCallback(pheromoneSpread_scale, XmNvalueChangedCallback,
pheromoneSpread_callback, NULL);

    pheromoneDrop_scale = XtVaCreateManagedWidget("pheromoneDropScale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, "Pheromone Drop
Level", 21,
            XmNmaximum, 1000,
            XmNminimum, 0,
            XmNscaleMultiple, 100,
            XmNvalue, PHEROMONE_STRENGTH,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 35,
            XmNy, WINHEIGHT - 380,
```

```
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
    XtAddCallback(pheromoneDrop_scale, XmNvalueChangedCallback,
pheromoneDrop_callback, NULL);

    maxTurns_scale = XtVaCreateManagedWidget("maxTurnsScale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, "Max Number of
Turns", 21,
            XmNmaximum, 10000,
            XmNminimum, 0,
            XmNscaleMultiple, 500,
            XmNvalue, MAX_TURNS,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 35,
            XmNy, WINHEIGHT - 315,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
    XtAddCallback(maxTurns_scale, XmNvalueChangedCallback,
maxTurns_callback, NULL);

    numAnts_scale = XtVaCreateManagedWidget("numAntsScale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, " Number of Ants",
16,
            XmNmaximum, MAX_ANTS,
            XmNminimum, 1,
            XmNvalue, NUM_ANTS,
            XmNscaleMultiple, 1,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 48,
            XmNy, WINHEIGHT - 250,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7.
            NULL);
    XtAddCallback(numAnts_scale, XmNvalueChangedCallback,
numAnts_callback, NULL);

    speed_scale = XtVaCreateManagedWidget("speedScale",
            xmScaleWidgetClass, bboard,
            XmNtraversalOn, FALSE,
            XtVaTypedArg, XmNtitleString, XmRString, "      Ant Speed",
16,
            XmNmaximum, 10,
            XmNminimum, 0,
            XmNvalue, 10,
            XmNshowValue, True,
            XmNorientation, XmHORIZONTAL,
            XmNprocessingDirection, XmMAX_ON_RIGHT,
            XmNx, 48,
            XmNy, WINHEIGHT - 185,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
```

```
    XtAddCallback(speed_scale, XmNvalueChangedCallback, speed_callback,
NULL);

    fileName_label = XtVaCreateManagedWidget("File Name:",
            xmLabelWidgetClass, bboard,
            XmNx, 10,
            XmNy, 20,
            XtVaTypedArg, XmNbackground, XmRString, "gray50", 7,
            NULL);
    fileName_text = XtVaCreateManagedWidget("filetext",
            xmTextFieldWidgetClass, bboard,
            XmNx, 80,
            XmNy, 15,
            XmNcolumns, 13,
            XmNmaxLength, 13,
            XmNvalue, fileName,
            XtVaTypedArg, XmNbackground, XmRString, "gray55", 7,
            NULL);
   XtAddCallback(fileName_text, XmNvalueChangedCallback, file_callback,
NULL);

    for ( i = 0; i < NUM_VALS; i++ ) {
      labels_text[i] = XtVaCreateManagedWidget(labels[i],
                        xmLabelWidgetClass, bboard,
                        XmNx, 10,
                        XmNy, 60 + 20*i,
                        XmNwidth, 100,
                        XmNalignment, XmALIGNMENT_END,
                        XtVaTypedArg, XmNbackground, XmRString,
"gray50", 7,
                        NULL);
      vals_text[i] = XtVaCreateManagedWidget(vals[i],
                        xmLabelWidgetClass, bboard,
                        XmNx, 120,
                        XmNy, 60 + 20*i,
                        XtVaTypedArg, XmNbackground, XmRString,
"gray50", 7,
                        NULL);
    }

    draw_frame = XtVaCreateManagedWidget("draw_frame",
            xmFrameWidgetClass, bboard,
            XmNx, 210,
            XmNy, 10,
            XmNshadowType, XmSHADOW_OUT,
            XmNshadowThickness, 4,
            XtVaTypedArg, XmNbackground, XmRString, "gray60", 7,
            NULL);

    draw_area = XtVaCreateManagedWidget("draw_area",
            xmDrawingAreaWidgetClass, draw_frame,
            XmNwidth, MAPX*PIXELS,
            XmNheight, MAPY*PIXELS,
      NULL);
   XtAddCallback(draw_area, XmNexposeCallback, drawArea_callback, NULL);

    window = XtWindow(draw_area);
    display = XtDisplay(draw_area);
```

```
    gc = XCreateGC( display, RootWindowOfScreen(XtScreen(draw_area)), 0,
NULL);

    pixmap = XCreatePixmap( XtDisplay(draw_area), RootWindowOf-
Screen(XtScreen(draw_area)),
                                            MAPX*PIXELS, MAPY*PIXELS,
                                            DefaultDepthOf-
Screen(XtScreen(draw_area)));

    XtRealizeWidget(toplevel);
}

void exit_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmPushButtonCallbackStruct *cbs;
{
    ERROR( E_DONE );
}

void execute_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
{
    if ( running ) {
      running = 0;
      changeState();
    } else {
      if ( !didRun ) {
          lastFoodInNest = 0;
          setUpRun();
          setVal( 0, 0 );
          setVal( 2, 0 );
          setVal( 3, NUM_FOOD );
          setVal( 4, 0 );
      }
      if (( legalFile ) && ( legalMap )) {
          didRun = 1;
          running = 1;
          changeState();
          execute(person);
          changeState();
      }
    }
}

void mapNum_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    int i, j;

    lastFoodInNest = 0;

    MAP_VERSION = cbs->value;

    if ( readMap() ) {
```

```
      legalMap = 1;
      if ( running ) {

          lastFoodInNest = food_in_nest = 0;
          setVal( 3, NUM_FOOD );

          for ( j = 0; j < MAPY; j++ )
            for ( i = 0; i < MAPX; i++ ) {
                if ( map[ i ][ j ].object != DEAD_ANT )
                  map[ i ][ j ].object = MAP[ i ][ j ].object;
                map[ i ][ j ].food = MAP[ i ][ j ].food;
            }
          for ( i = 0; i < NUM_FOOD; i++ ) {
            foodId[ i ].weight = FOODID[ i ].weight;
            foodId[ i ].numAntsLifting = 0;
            foodId[ i ].found = 0;
          }
          for ( i = 0; i < NUM_ANTS; i++ )
            ants[ i ].foodCarried = NO_FOOD;
      } else
          initialMapDraw();
    }
}

void pheromoneSpread_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    PHEROMONE_SPREAD = cbs->value;
}

void pheromoneDrop_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    PHEROMONE_STRENGTH = cbs->value;
}

void maxTurns_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    MAX_TURNS = cbs->value;

    changeState();
}

void numAnts_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    NUM_ANTS = cbs->value;
}

void speed_callback(widget, client_data, cbs)
```

```
Widget widget;
XtPointer client_data;
XmScaleCallbackStruct *cbs;
{
    antSpeed = 10 - cbs->value;
}

void file_callback(widget, client_data, cbs)
Widget widget;
XtPointer client_data;
XmTextVerifyCallbackStruct *cbs;
{
    XmUpdateDisplay(toplevel);

    strcpy(fileName, XmTextFieldGetString(widget));
    savedFile( fileName );
}

void drawArea_callback(widget, data, cbs)
Widget widget;
XtPointer data;
XmDrawingAreaCallbackStruct *cbs;
{
    XEvent      *event = cbs->event;
    Display     *dpy = event->xany.display;

    XCopyArea (XtDisplay(draw_area), pixmap, XtWindow(draw_area),
                gc, 0, 0, MAPX*PIXELS, MAPY*PIXELS, 0, 0);
}

void mapDump()
{
    int i, j, color;
    char colorStr[20];
    XEvent event;

    set_color (draw_area, "white");
   XFillRectangle( XtDisplay(draw_area), pixmap, gc, 0, 0, MAPX*PIXELS,
MAPY*PIXELS );

    for ( i = 0; i < MAPY; i++ ) {
      for ( j = 0; j < MAPX; j++ ) {
          if (map[j][i].pheromone) {
            if ( PHEROMONE_STRENGTH != 0 )
                if ( PHEROMONE_SPREAD != 0 )
                  color = 97 - ( map[j][i].pheromone / (
PHEROMONE_STRENGTH / 10 ) );
                else
                  color = 97 - ( map[j][i].pheromone / PHEROMONE_STRENGTH
);
            else
                color = 97 - ( map[j][i].pheromone / 97 );
            if (( color < 45 ) || ( color > 97 ))
                color = 45;
            sprintf(colorStr, "gray%d", color);
            set_color (draw_area, colorStr);
            XFillRectangle( XtDisplay(draw_area), pixmap, gc, loc(j,i) );
          }
          switch ( map[j][i].object ) {
```

```
            case NOTHING :
                break;
            case DEAD_ANT :
                set_color (draw_area, "OrangeRed2");
                XDrawLine( XtDisplay(draw_area), pixmap, gc, j*PIXELS,
i*PIXELS,
                                                    (j+1)*PIXELS-1,
(i+1)*PIXELS-1 );
               XDrawLine( XtDisplay(draw_area), pixmap, gc, (j+1)*PIXELS-
1, i*PIXELS,
                                                     j*PIXELS, (i+1)*PIXELS-
1 );
                break;
            case WALL :
                set_color (draw_area, "red");
              XFillRectangle( XtDisplay(draw_area), pixmap, gc, loc(j,i)
);
                break;
            case WATER :
                set_color (draw_area, "blue");
              XFillRectangle( XtDisplay(draw_area), pixmap, gc, loc(j,i)
);
                break;
            case NEST :
                set_color (draw_area, "goldenrod2");
               XFillArc( XtDisplay(draw_area), pixmap, gc, loc(j,i), 0,
64*360 );
          }
          if ( map[j][i].food != NO_FOOD ) {
                set_color (draw_area, "green");
                XFillRectangle( XtDisplay(draw_area), pixmap, gc,
small_loc(j,i) );
          }
      }
    }

    for ( i = 0; i < NUM_ANTS; i++ ) {
      if ( ants[i].alive ) {
          if ( ants[i].foodCarried != NO_FOOD )
            set_color (draw_area, "oliveDrab");
          else
            set_color (draw_area, "brown");
          drawAnt( ants[i].loc.x, ants[i].loc.y, ants[i].dir );
      }
    }

    XCopyArea (XtDisplay(draw_area), pixmap, XtWindow(draw_area),
                gc, 0, 0, MAPX*PIXELS, MAPY*PIXELS, 0, 0);

   setVal( 0, findFitness( NUM_FOOD - food_in_nest, foodNotFound, ( loops
> turns/NUM_ANTS ) ? loops : turns/NUM_ANTS , numOfNodes ));

    setVal( 2, ( loops > turns/NUM_ANTS ) ? loops : turns/NUM_ANTS );

    if ( lastFoodInNest != food_in_nest ) {
      lastFoodInNest = food_in_nest;
      setVal( 3, NUM_FOOD   food_in_nest );
    }
```

```
    setVal( 4, foodNotFound );

    XmUpdateDisplay(toplevel);

    for ( i = 0; i < XQLength(XtDisplay(toplevel)); i++ ) {
      XtAppNextEvent(app, &event);
      XtDispatchEvent(&event);
    }
}

main(argc, argv)
int argc;
char *argv[];
{
    fileName[0] = 0;

    status = 0;

    antSpeed = 0;
    MAP_VERSION = 1;
    NUM_ANTS = 20;
    MAX_TURNS = 2500;
    PHEROMONE_STRENGTH = 100;
    PHEROMONE_SPREAD = 40;
    FOOD_LEFT_WEIGHT = 10000000;
    FOOD_UNFOUND_WEIGHT = 100000;
    TIME_WEIGHT = 1;
    NODE_WEIGHT = 10000;
    NODE_GROUP = 50;
    WAIT_ODDS = 50;

    ants = (AntsType) alloVector( MAX_ANTS, sizeof(AntType));

    if ( argc == 2 )
      strcpy( fileName, argv[1] );

    makeWidgets(argc, argv);

    if ( readMap() ) {
      legalMap = 1;
      initialMapDraw();
    }

    if ( argc == 2 )
      savedFile( argv[1] );

    changeState();

    XtAppMainLoop(app);

    free( ants );
}
```

```
/*
** dump.c
**
** routines to print out the tree
**
*/

#include <stdio.h>
#include <time.h>
#include "params.h"
#include "types.h"
#include "commands.h"
#include "dir.h"
#include "error.h"

FILE *fpo;
FILE *fpi;
extern int firstUsed;
extern int itteration;
extern int NUM_FOOD;

extern char *CommandText[];

treeDump( tree )
TreePtr tree;
{
    fpo = stdout;
    treeDumpRecur( tree, 0 );
}

treeDumpRecur( tree, depth )
TreePtr tree;
int   depth;
{
    int i;

    if ( tree == NULL_TREE )
      return;

    for (i = 0; i < depth; i++)
      fprintf(fpo, "    ");

    fprintf(fpo, "%d\n", tree->type );

    treeDumpRecur( tree->left, depth+1 );
    treeDumpRecur( tree->right, depth+1 );
}

fileStringTreeDump( person )
ProgramType person;
{
    char filename[70];
    char filecom[150];
    char ittnum[10];
    TreePtr tree;

    tree = person.tree;

    sprintf( filename, "%s%02d.%06d", TEST_FILE, VERSION, itteration );
```

```
    if (( fpo = fopen( filename, "w")) == 0 ) {
      printf("error opening file %s\n", filename);
      ERROR( E_OPEN_RESULT );
    }

    doStats( person );
    stringTreeDumpRecur( tree, 0, 1 );
/*treeDumpRecur(tree,0);*/
    fclose( fpo );

    fpo = stdout;
    doStats( person );
}

stringTreeDump( person )
ProgramType person;
{
    TreePtr tree;

    tree = person.tree;

    fpo = stdout;
    doStats( person );
    stringTreeDumpRecur( tree, 0, 1 );
}

stringTreeDumpRecur( tree, depth, all)
TreePtr tree;
int   depth;
int   all;
{
    int i;

    if ( tree == NULL_TREE ) {
      fprintf(fpo, "NULL\n");
      ERROR( E_NULL_TREE );
    }

    if ( isLeaf( tree->type )) {
      doTabs( depth );
      fprintf(fpo, "%s\n", CommandText[ tree->type ] );

    } else if ( isNonLeaf( tree->type )) {

      if ( tree->type == PROC2 ) {
          stringTreeDumpRecur( tree->left, depth, all );
          stringTreeDumpRecur( tree->right, depth, all );

      } else
          dumpIfElse( CommandText[ tree->type ], tree, depth, all );

    } else {
      printf("illegalType - %d\n", tree->type );
      ERROR( E_ILLEGAL_INSTRUCTION );
    }
}

dumpIfElse( str, tree, depth, all )
```

```
char *str;
TreePtr tree;
int depth;
int all;
{
    doTabs( depth );
    depth++;
    fprintf(fpo, "%s\n", str );
    if ( all ) {
      stringTreeDumpRecur( tree->left, depth, all );
      doTabs( depth-1 );
      fprintf(fpo, "%s\n", CommandText[ ELSE ] );
      stringTreeDumpRecur( tree->right, depth, all );
    }
}

doTabs( depth )
int depth;
{
    int i;

    for (i = 0; i < depth; i++)
      fprintf(fpo, CommandText[ TAB ] );
}

doStats( person )
ProgramType person;
{
    char *stime;
    time_t tim;

    tim = time(0);
    stime = ctime(&tim);
    stime[19] = '\0';

    if ( firstUsed )
      printf("First Fitness = %9d,\n\tNumber of nodes = %3d, Number of
Turns = %4d,\n\tFood Left = %2d, Food Not Found = %2d\n",
                  person.fitness, person.numNodes, person.numTurns,
                  person.foodLeft, person.foodNotFound );
    else
      fprintf(fpo, "Iteration = %6d, Time = %s, Fitness = %9d,\n\tNumber
of Nodes = %3d, Number of Turns = %4d,\n\tFood Left = %2d, Food Not Found
= %2d\n",
                  itteration, stime, person.fitness, person.numNodes,
person.numTurns,
                  person.foodLeft, person.foodNotFound);

/*    free( stime );
*/
}

fileParamDump()
{
    int i;
    TreeType tree;

    fprintf(fpo, "POPULATION_SIZE          = %d\n", POPULATION_SIZE);
    fprintf(fpo, "TOURNAMENT_SIZE          = %d\n", TOURNAMENT_SIZE);
```

```
    fprintf(fpo, "FOOD_LEFT_WEIGHT          = %d\n", FOOD_LEFT_WEIGHT);
    fprintf(fpo, "FOOD_UNFOUND_WEIGHT       = %d\n", FOOD_UNFOUND_WEIGHT);
    fprintf(fpo, "TIME_WEIGHT               = %d\n", TIME_WEIGHT);
    fprintf(fpo, "NODE_WEIGHT               = %d\n", NODE_WEIGHT);
    fprintf(fpo, "NODE_GROUP                = %d\n", NODE_GROUP);
    fprintf(fpo, "LEAF_ODDS                 = %d\n", LEAF_ODDS);
    fprintf(fpo, "NON_LEAF_ODDS             = %d\n", NON_LEAF_ODDS);
    fprintf(fpo, "MUTATE_ODDS               = %d\n", MUTATE_ODDS);
    fprintf(fpo, "PERCENT_GREEDY_MUTATE     = %d\n",
PERCENT_GREEDY_MUTATE);
    fprintf(fpo, "MAX_TURNS                 = %d\n", MAX_TURNS);
    fprintf(fpo, "NUM_ANTS                  = %d\n", NUM_ANTS);
    fprintf(fpo, "GOAL_FITNESS              = %d\n", GOAL_FITNESS);
    fprintf(fpo, "POP_SEED                  = %d\n", POP_SEED);
    fprintf(fpo, "INTERP_SEED               = %d\n", INTERP_SEED);
    fprintf(fpo, "PHEROMONE_STRENGTH        = %d\n", PHEROMONE_STRENGTH);
    fprintf(fpo, "PHEROMONE_SPREAD          = %d\n", PHEROMONE_SPREAD);
    fprintf(fpo, "WAIT_ODDS                 = %d\n", WAIT_ODDS);
    fprintf(fpo, "AUTO_FOOD_DROP            = %d\n", AUTO_FOOD_DROP);
    fprintf(fpo, "DEBUG                     = %d\n", DEBUG);
    fprintf(fpo, "SHOW_TIME                 = %d\n", SHOW_TIME);
    fprintf(fpo, "DUMP                      = %d\n", DUMP);
    fprintf(fpo, "MAX_ITERATIONS            = %d\n", MAX_ITERATIONS);
    fprintf(fpo, "RESTORE_ITERATION         = %d\n", RESTORE_ITERATION);
    fprintf(fpo, "RESTORE_VERSION           = %d\n", RESTORE_VERSION);
    fprintf(fpo, "RESTORE_AMOUNT            = %d\n", RESTORE_AMOUNT);
    fprintf(fpo, "NUM_FOOD                  = %d\n", NUM_FOOD);
    fprintf(fpo, "INSTRUCTION_VERSION       = %d\n", INSTRUCTION_VERSION);
    fprintf(fpo, "MAP_VERSION               = %d\n", MAP_VERSION);
    fprintf(fpo, "VERSION                   = %d\n", VERSION);

    fprintf(fpo, "\n\nInstruction Set:\n\n");

    tree.left = NULL;
    tree.right = NULL;

    for ( i = 0; i < numLeafs; i++ ) {
      tree.type = Leafs[i];
      stringTreeDumpRecur( &tree, 0, 0 );
    }

    fprintf(fpo, "\n");

    for ( i = 0; i < numNonLeafs; i++ ) {
      tree.type = NonLeafs[i];
      if ( tree.type == PROC2 )
          fprintf(fpo, "PROC2\n");
      else
          stringTreeDumpRecur( &tree, 0, 0 );
    }
}

paramDump()
{
    char filename[70];

    sprintf( filename, "%s%02d", TEST_FILE, VERSION);

    if ( fpi = fopen( filename, "r" )) {
```

```
      printf("\nFile %s already exists.\n\n", filename);
      fclose( fpi );
      ERROR( E_NOT_NEW_VERSION );
    }

    if (( fpo = fopen( filename, "w")) == 0 ) {
      printf("error opening file %s\n", filename);
      ERROR( E_OPEN_RESULT );
    }

    fileParamDump();

    fclose( fpo );

    fpo = stdout;

    fileParamDump();

    printf("\n\n\n");
}
```

```
/*
**
** error.c
**
*/

#include "error.h"

void error( file, line, errorNum )
char *file;
int line;
ErrorCode errorNum;
{
    if ( errorNum == E_DONE ) {
      exit( 0 );
    }

    printf( "%s: line %d : ", file, line );

    switch ( errorNum ) {
      case E_NO_PARAM_FILE_SPECIFIED :
          printf( "Must specify param file number" );
          break;
      case E_OPEN_PARAM_FILE :
          printf( "Cannot open param file" );
          break;
      case E_ALLOC :
          printf( "Error allocating memory" );
          break;
      case E_MAP :
          printf( "Bad map file" );
          break;
      case E_TWO_NEST :
          printf( "Two nest areas defined in map file" );
          break;
      case E_ILLEGAL_INSTRUCTION :
          printf( "Illegal command code" );
          break;
      case E_OPEN_INSTRUCTION_FILE :
          printf( "Cant open instruction file" );
          break;
      case E_INSTRUCTION_SETTING :
          printf( "Illegal instruction file" );
          break;
      case E_OPEN_RESULT :
          printf( "Error opening results file" );
          break;
      case E_NULL_TREE :
          printf( "Illegal NULL value for tree" );
          break;
      case E_NOT_NEW_VERSION :
          printf( "Files already exist for VERSION" );
          break;
      case E_FIRST_FILE :
          printf( "Bad first file" );
          break;
      case E_DIRECTION :
          printf( "Illegal direction in intepreter" );
          break;
```

```
        case E_MAP_FILE :
            printf( "Bad character in map file" );
            break;
        case E_WRITE_SAVE_FILE :
            printf( "Error writing save file" );
            break;
        case E_GET_SAVE_FILE :
            printf( "Error reading save file" );
            break;
        case E_SAVE_FILE :
            printf( "Error in save file" );
            break;
        case E_SAME_VERSION :
            printf( "VERSION and RESTORE_VERSION cannot be the same" );
            break;
        case E_LEX :
            printf( "Error in lex" );
            break;
        case E_BAD_INPUT_SPACING :
           printf( "Bad number of spaces for tab in input file.  Check for
spaces before tabs" );
            break;
        default :
            printf( "Unknown error code" );
            break;
    }

    printf( "\n\n" );

    exit( 1 );
}
```

```
/*
** execute.c
**
** the execute() function is the interp function for the display
**
*/

#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

/* dir:   0 = N, 1 = E, 2 = S, 3 = W */

extern int antSpeed;

MapType map[ MAPX ][ MAPY ];
extern int loops;
extern int running;

extern int turns;
extern int food_in_nest;
extern int nodes;

extern TreePtratree;
extern int ant;
extern int NUM_FOOD;
extern MapType MAP[ MAPX ][ MAPY ];
extern AntsType ants;
extern FoodIDType FOODID[ MAX_FOOD ];
extern FoodIDType foodId[ MAX_FOOD ];

int foodNotFound;

void execute( person )
ProgramType person;
{
    int i,j;
    TreePtr tree;

    tree = person.tree;
    atree = tree;

    if ( tree == NULL_TREE )
      ERROR( E_NULL_TREE );

    mapDump();
    while ( running ) {
      if (!(( food_in_nest < NUM_FOOD ) && ( turns < MAX_TURNS*NUM_ANTS )
&& ( loops++ < MAX_TURNS ))) {
          running = 0;
          return;
      }
      for (i = 0; i < antSpeed*1000000; i++ );
      if ( PHEROMONE_SPREAD != 0 )
          spreadPheromone();
      setUpFollowFood();
      for ( ant = 0; ant < NUM_ANTS; ant++ )
          if ( ants[ ant ].alive )
```

```
            startAnt( tree );
        foodNotFound = 0;
        for ( i = 0; i < NUM_FOOD; i++ )
            if ( !foodId[ i ].found )
                foodNotFound++;
        mapDump();
    }
}
```

```
/*
** init.c
**
** create the initial population
**
*/

#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

int *hood;
int *pair;
PopType population;
extern AntsType ants;

extern int NUM_FOOD;

int newStmt()
{
    if ( ( lrand48() % ( LEAF_ODDS + NON_LEAF_ODDS ) ) < LEAF_ODDS )
      return( getLeafStmt() );
    else
      return( getNonLeafStmt() );
}

initialPopulation( start )
int start;
{
    int i;
    int nod, food, turn;

    dprintf("Start of init()\n");

    for ( i = start; i < POPULATION_SIZE; i++ ) {
      dprintf("Individual = %d, start of loop initialPopulation\n", i);

        population[i].tree = (TreePtr) allocMem( sizeof( TreeType ) );
        population[i].tree->type = getNonLeafStmt();
      population[i].tree->left = (TreePtr) allocMem( sizeof( TreeType ) );
      population[i].tree->right = (TreePtr) allocMem( sizeof( TreeType )
);
      population[i].tree->left->type = newStmt();
      population[i].tree->right->type = newStmt();

      stmt( population[ i ].tree->left );
      stmt( population[ i ].tree->right );

      dprintf("Individual = %d, before interp in initialPopulation\n",
i);

      interp( &population[ i ] );

      dprintf("Individual = %d, end of loop initialPopulation\n", i);
    }
}
```

```
stmt( tree )
TreePtrtree;
{
    if ( tree->type == PROC2 ) {
      dprintf("PROC2 in stmt\n");
      tree->left = (TreePtr) alloMem( sizeof( TreeType ) );
      tree->left->type = newStmt();
      stmt( tree->left );
      tree->right = (TreePtr) alloMem( sizeof( TreeType ) );
      tree->right->type = newStmt();
      stmt( tree->right );
    } else if ( isLeaf( tree->type )) {
      dprintf("Non terminal in stmt\n");
      tree->left = NULL_TREE;
      tree->right = NULL_TREE;
    } else if ( isNonLeaf( tree->type )) {
      dprintf("IF_statement in stmt\n");
      tree->left = (TreePtr) alloMem( sizeof( TreeType ) );
      tree->left->type = newStmt();
      stmt( tree->left );
      tree->right = (TreePtr) alloMem( sizeof( TreeType ) );
      tree->right->type = newStmt();
      stmt( tree->right );
    } else {
      printf("Illegal type: %d\n", tree->type );
      ERROR( E_ILLEGAL_INSTRUCTION );
    }
}

setUpVars()
{
    hood = (int *) alloVector( TOURNAMENT_SIZE, sizeof( int ));
    pair = (int *) alloVector( POPULATION_SIZE, sizeof( int ));
    population = (PopType) alloVector( POPULATION_SIZE, sizeof(Program-
Type));
    ants = (AntsType) alloVector( MAX_ANTS, sizeof(AntType));
}

findBest( bestIndividual )
int   *bestIndividual;
{
    int i;
    int bestFitness;

    *bestIndividual = 0;
    bestFitness = population[0].fitness;

    for ( i = 1; i < POPULATION_SIZE; i++ ) {
      if ( population[ i ].fitness < bestFitness ) {
         *bestIndividual = i;
         bestFitness = population[ i ].fitness;
      }
    }
}
```

```
/*
** interp.c
**
** the interp() function will execute one program in binary tree form and
**    evaluate its performance.  It then returns the fitness value.
**
*/

#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

/* dir:   0 = N, 1 = E, 2 = S, 3 = W */

extern int firstUsed;

MapType map[ MAPX ][ MAPY ];
DIRS dir1, dir2, tmpDir;
int ant;
int ph1, ph2, dif1, dif2;
int i;
int loops;
int turns;
int nodes;
int pLevel, maxPlevel;
DIRS pDir;
TreePtratree;
int food_in_nest;

extern int NUM_FOOD;
extern MapType MAP[ MAPX ][ MAPY ];
extern int NESTX;
extern int NESTY;
AntsType ants;
extern FoodIDType FOODID[ MAX_FOOD ];
FoodIDType foodId[ MAX_FOOD ];

void turn_left()
{
    ants[ant].dir = ants[ant].dir - 1;
    if ( ants[ant].dir < 0 )
      ants[ant].dir = 3;
}

void turn_right()
{
    ants[ant].dir = (ants[ant].dir + 1) % 4;
}

void pick_up_food()
{
    if ( ants[ ant ].foodCarried == NO_FOOD )
      if ( map[ ants[ant].loc.x ][ ants[ant].loc.y ].food != NO_FOOD ) {
          ants[ ant ].foodCarried = map[ ants[ant].loc.x ][
ants[ant].loc.y ].food;
          foodId[ ants[ ant ].foodCarried ].found = 1;
          foodId[ ants[ ant ].foodCarried ].numAntsLifting++;
```

```
        if ( foodId[ ants[ ant ].foodCarried ].numAntsLifting == foodId[
ants[ ant ].foodCarried ].weight ) {
            foodId[ ants[ant].foodCarried ].foundThisTurn = 1;
            foodId[ ants[ant].foodCarried ].moved = 1;
            foodId[ ants[ant].foodCarried ].leader = ant;
            map[ ants[ant].loc.x ][ ants[ant].loc.y ].food = NO_FOOD;
          }
      }
}

void drop_food()
{
   if ((( map[ ants[ant].loc.x ][ ants[ant].loc.y ].object == NOTHING ) ||
                        ( map[ ants[ant].loc.x ][ ants[ant].loc.y
].object == NEST ) ||
                        ( map[ ants[ant].loc.x ][ ants[ant].loc.y
].object == DEAD_ANT )) &&
                        (( map[ ants[ant].loc.x ][ ants[ant].loc.y ].food
== NO_FOOD ) ||
                         ( map[ ants[ant].loc.x ][ ants[ant].loc.y ].food
== ants[ant].foodCarried )) &&
                        ( ants[ant].foodCarried != NO_FOOD )) {
      if ( map[ ants[ant].loc.x ][ants[ant].loc.y ].object == NEST ) {
            food_in_nest++;
            for ( i = 0; i < NUM_ANTS; i++ )
               if (( ants[i].foodCarried == ants[ ant ].foodCarried ) &&
( i != ant ))
                  ants[i].foodCarried = NO_FOOD;
      } else {
            foodId[ ants[ ant ].foodCarried ].numAntsLifting--;
            map[ ants[ant].loc.x ][ ants[ant].loc.y ].food = ants[ ant
].foodCarried;
      }
      ants[ ant ].foodCarried = NO_FOOD;
    }
}

int can_move()
{
    if ( ants[ ant ].foodCarried != NO_FOOD )
      if (( foodId[ ants[ ant ].foodCarried ].numAntsLifting < foodId[
ants[ ant ].foodCarried ].weight ) ||
            (( foodId[ ants[ ant ].foodCarried ].numAntsLifting > 1 ) &&
             ( foodId[ ants[ ant ].foodCarried ].foundThisTurn )))
          return( 0 );
    return( 1 );
}

int move_forward()
{
    if ( !can_move() )
      return( 0 );

    if ( ants[ ant ].foodCarried != NO_FOOD ) {
      foodId[ ants[ ant ].foodCarried ].moved = 1;
      foodId[ ants[ ant ].foodCarried ].leader = ant;
    }

    switch ( ants[ant].dir ) {
```

```
      case NORTH :
          if (( ants[ant].loc.y > 0 ) &&( map[ ants[ant].loc.x ][
ants[ant].loc.y-1 ].object != WALL ) &&
                                        ( map[ ants[ant].loc.x ][
ants[ant].loc.y-1 ].object != WATER ))
            ants[ant].loc.y--;
          else
            return( 0 );
          break;
      case EAST :
          if (( ants[ant].loc.x < MAPX-1 ) &&( map[ ants[ant].loc.x+1 ][
ants[ant].loc.y ].object != WALL ) &&
                                        ( map[ ants[ant].loc.x+1 ][
ants[ant].loc.y ].object != WATER ))
            ants[ant].loc.x++;
          else
            return( 0 );
          break;
      case SOUTH :
          if (( ants[ant].loc.y < MAPY-1 ) &&( map[ ants[ant].loc.x ][
ants[ant].loc.y+1 ].object != WALL ) &&
                                        ( map[ ants[ant].loc.x ][
ants[ant].loc.y+1 ].object != WATER ))
            ants[ant].loc.y++;
          else
            return( 0 );
          break;
      case WEST :
          if (( ants[ant].loc.x > 0 ) &&( map[ ants[ant].loc.x-1 ][
ants[ant].loc.y ].object != WALL ) &&
                                        ( map[ ants[ant].loc.x-1 ][
ants[ant].loc.y ].object != WATER ))
            ants[ant].loc.x--;
          else
            return( 0 );
          break;
      default :
          printf("Illegal direction in move_forward(): %d\n", ants[ ant
].dir );
          ERROR( E_DIRECTION );
    }

    if (( map[ ants[ant].loc.x ][ants[ant].loc.y ].object == NEST ) && (
AUTO_FOOD_DROP ))
      drop_food();

    turns++;

    return( 1 );
}

int move_to_nest()
{
   if ( abs( ants[ ant ].loc.x - NESTX ) > abs( ants[ ant ].loc.y - NESTY
) )
      if ( ants[ ant ].loc.x > NESTX )
          ants[ ant ].dir = WEST;
      else
          ants[ ant ].dir = EAST;
```

```
    else
      if ( ants[ ant ].loc.y > NESTY )
          ants[ ant ].dir = NORTH;
      else
          ants[ ant ].dir = SOUTH;

    return( move_forward() );
}

int what_ahead( what )
OBJECT_TYPE what;
{
    switch ( ants[ant].dir ) {
      case NORTH :
          if ( ants[ant].loc.y > 0 )
            switch ( what ) {
                case OBJECT    : return( map[ ants[ant].loc.x ][
ants[ant].loc.y-1 ].object );
                                 break;
                case FOOD      : return( map[ ants[ant].loc.x ][
ants[ant].loc.y-1 ].food != NO_FOOD );
                                 break;
                case PHEROMONE : return( map[ ants[ant].loc.x ][
ants[ant].loc.y-1 ].pheromone );
                                 break;
                default   : printf( "Illegal type = %d in what_ahead()\n",
what );
            }
          break;
      case EAST :
          if ( ants[ant].loc.x < MAPX-1 )
            switch ( what ) {
                case OBJECT    : return ( map[ ants[ant].loc.x+1 ][
ants[ant].loc.y ].object );
                                 break;
                case FOOD      : return ( map[ ants[ant].loc.x+1 ][
ants[ant].loc.y ].food != NO_FOOD );
                                 break;
                case PHEROMONE : return ( map[ ants[ant].loc.x+1 ][
ants[ant].loc.y ].pheromone );
                                 break;
                default   : printf( "Illegal type = %d in what_ahead()\n",
what );
            }
          break;
      case SOUTH :
          if ( ants[ant].loc.y < MAPY-1 )
            switch ( what ) {
                case OBJECT    : return ( map[ ants[ant].loc.x ][
ants[ant].loc.y+1 ].object );
                                 break;
                case FOOD      : return ( map[ ants[ant].loc.x ][
ants[ant].loc.y+1 ].food != NO_FOOD );
                                 break;
                case PHEROMONE : return ( map[ ants[ant].loc.x ][
ants[ant].loc.y+1 ].pheromone );
                                 break;
                default   : printf( "Illegal type = %d in what_ahead()\n",
what );
```

```
            }
          break;
      case WEST :
          if ( ants[ant].loc.x > 0 )
            switch ( what ) {
                case OBJECT    : return ( map[ ants[ant].loc.x-1 ][
ants[ant].loc.y ].object );
                                 break;
                case FOOD      : return ( map[ ants[ant].loc.x-1 ][
ants[ant].loc.y ].food != NO_FOOD );
                                 break;
                case PHEROMONE : return ( map[ ants[ant].loc.x-1 ][
ants[ant].loc.y ].pheromone );
                                 break;
                default   : printf( "Illegal type = %d in what_ahead()\n",
what );
            }
          break;
      default :
          printf("Illegal direction in what_ahead(): %d\n", ants[ ant
].dir );
          ERROR( E_DIRECTION );
    }
    return( NOTHING );
}

int move_into_water()
{
    if ( !can_move() )
      return( 0 );

    if ( what_ahead( OBJECT ) == WATER ) {
      switch( ants[ ant ].dir ) {
          case NORTH :
            map[ ants[ant].loc.x ][ ants[ant].loc.y-1 ].object =
DEAD_ANT;
            break;
          case EAST :
            map[ ants[ant].loc.x+1 ][ ants[ant].loc.y ].object =
DEAD_ANT;
            break;
          case SOUTH :
            map[ ants[ant].loc.x ][ ants[ant].loc.y+1 ].object =
DEAD_ANT;
            break;
          case WEST :
            map[ ants[ant].loc.x-1 ][ ants[ant].loc.y ].object =
DEAD_ANT;
            break;
      }

      move_forward();

      ants[ ant ].alive = 0;

      if ( ants[ant].foodCarried != NO_FOOD ) {
          foodId[ ants[ant].foodCarried ].moved = 1;
          foodId[ ants[ant].foodCarried ].leader = ant;
          drop_food();
```

```
      }
      return( 1 );
    }
    else
      return( 0 );
}

turn_to_adjacent_object( object )
OBJECT_TYPE object;
{
    if ( !what_ahead( object ) ) {
      turn_right();
      if ( !what_ahead( object ) ) {
          turn_left();
          turn_left();
          if ( !what_ahead( object ) ) {
            turn_left();
            if ( !what_ahead( object ) ) {
                turn_right();
                turn_right();
            }
          }
      }
    }
}

void statement( tree )
TreePtr tree;
{
    int val2;
    int cond;
    int i;

    if (( tree == NULL_TREE ) || ( ! ants[ ant ].alive ))
      return;

    switch ( tree->type ) {
      case PROC2 :
          statement( tree->left );
          statement( tree->right );
          break;
      case TURN_LEFT :
          turn_left();
          break;
      case TURN_RIGHT :
          turn_right();
          break;
      case MOVE_FORWARD :
          move_forward();
          break;
      case MOVE_RANDOM :
          ants[ ant ].dir = lrand48() % 4;
          move_forward();
          move_forward();
          break;
      case MOVE_QUASI_RANDOM :
            /* 50% straight, 20% left or right, 10% behind */
          cond = lrand48() % 10;
          if ( cond < 2 )
```

```
            turn_left();
          else if ( cond < 4 )
            turn_right();
          else if ( cond < 5 ) {
            turn_right();
            turn_right();
          }
          move_forward();
          move_forward();
          break;
        case MOVE_TO_NEST :
          move_to_nest();
          break;
        case MOVE_TO_NEST_THROUGH_WATER :
          if ( !move_to_nest() )
            move_into_water();
          break;
        case MOVE_INTO_WATER :
          move_into_water();
          break;
        case PICK_UP_FOOD :
          pick_up_food();
          break;
        case DROP_FOOD :
          drop_food();
          break;
        case RELEASE_PHEROMONE :
          map[ ants[ant].loc.x ][ ants[ant].loc.y ].pheromone +=
PHEROMONE_STRENGTH;
          break;
        case ESCAPE_PHEROMONE :
          drop_food();
          while (( move_forward()) && ( what_ahead( PHEROMONE )))
            move_forward();
          move_forward();
          move_forward();
          break;
        case NOOP :
          break;
        case IF_FOOD_AHEAD :
          if ( what_ahead( FOOD ) )
            statement( tree->left );
          else
            statement( tree->right );
          break;
        case IF_FOOD_LEFT :
          turn_left();
          cond = what_ahead( FOOD );
          turn_right();
          if ( cond )
            statement( tree->left );
          else
            statement( tree->right );
          break;
        case IF_FOOD_RIGHT :
          turn_right();
          cond = what_ahead( FOOD );
          turn_left();
          if ( cond )
```

```
      statement( tree->left );
    else
      statement( tree->right );
    break;
case IF_FOOD_ADJACENT :
    cond = 0;
    for ( i = 0; i < 4; i++ ) {
      if ( what_ahead( FOOD ) )
          cond = 1;
      turn_right();
    }
    if ( cond )
      statement( tree->left );
      else
          statement( tree->right );
    break;
case IF_MOVE_TO_ADJACENT_FOOD :
    turn_to_adjacent_object( FOOD );
    if ( what_ahead( FOOD ) )
    {
      move_forward();
      statement( tree->left );
    }
      else
          statement( tree->right );
      break;
case IF_MOVE_TO_ADJACENT_PHEROMONE :
    maxPlevel = 0;

    pLevel = what_ahead( PHEROMONE );
    if ( pLevel > maxPlevel ) {
      maxPlevel = pLevel;
      pDir = 0;
    }

    turn_right();
    pLevel = what_ahead( PHEROMONE );
    if ( pLevel > maxPlevel ) {
      maxPlevel = pLevel;
      pDir = 1;
    }

    turn_left();
    turn_left();
    pLevel = what_ahead( PHEROMONE );
    if ( pLevel > maxPlevel ) {
      maxPlevel = pLevel;
      pDir = 3;
    }

    turn_left();
    pLevel = what_ahead( PHEROMONE );
    if ( pLevel > maxPlevel ) {
      maxPlevel = pLevel;
      pDir = 2;
    }

    turn_right();
    turn_right();
```

```
          if ( maxPlevel > 0 )
          {
            for ( i = 0; i < pDir; i++ )
                turn_right();
            move_forward();
                statement( tree->left );
          }
            else
                statement( tree->right );
          break;
      case IF_MOVE_TO_AWAY_PHEROMONE :
          if ( ants[ ant ].loc.x < NESTX )
            dir1 = WEST;
          else
            dir1 = EAST;

          tmpDir = ants[ ant ].dir;
          ants[ ant ].dir = dir1;
          ph1 = what_ahead( PHEROMONE );

          if ( ants[ ant ].loc.y < NESTY )
            dir2 = NORTH;
          else
            dir2 = SOUTH;

          ants[ ant ].dir = dir2;
          ph2 = what_ahead( PHEROMONE );

          if ( ph1 || ph2 ) {
/*
FOR SLIGHTLY RANDOMIZING PHEROMONE FOLLOWING
            ph1 += (float) ph1 * drand48() * .2;
            ph2 += (float) ph2 * drand48() * .2;
*/
            if ( ph1 > ph2 )
                ants[ ant ].dir = dir1;
            else if ( ph2 > ph1 )
                ants[ ant ].dir = dir2;
            else if (( tmpDir == dir1 ) || ( tmpDir == dir2 ))
            {
                ants[ ant ].dir = tmpDir;
            }
            else if (( tmpDir % 2 ) == ( dir1 % 2 ))
                 ants[ ant ].dir = dir2;
            else
                 ants[ ant ].dir = dir1;

            move_forward();
                statement( tree->left );
          } else {
            ants[ ant ].dir = tmpDir;
                statement( tree->right );
          }
          break;
      case IF_MOVE_TO_DEAD_ANT :
         if ( map[ ants[ant].loc.x ][ ants[ant].loc.y ].object != DEAD_ANT
)
            turn_to_adjacent_object( DEAD_ANT );
```

```
            if ( what_ahead( OBJECT ) == DEAD_ANT ) {
              move_forward();
                  statement( tree->left );
            } else {
                  statement( tree->right );
            }
            break;
        case IF_MOVE_TO_NEST :
            if ( move_to_nest() )
                  statement( tree->left );
            else
              statement( tree->right );
            break;
        case IF_MOVE_TO_NEST_THROUGH_WATER :
            if ( !move_to_nest() )
              if ( !move_into_water() )
                  statement( tree->left );
              else
                  statement( tree->right );
            else
              statement( tree->right );
            break;
        case IF_WATER_AHEAD :
            if ( what_ahead( OBJECT ) == WATER )
              statement( tree->left );
              else
                  statement( tree->right );
            break;
        case IF_OBSTACLE_AHEAD :
            if ( what_ahead( OBJECT ) == WALL )
              statement( tree->left );
              else
                  statement( tree->right );
            break;
        case IF_FOOD_HERE :
           if ( map[ ants[ant].loc.x ][ ants[ant].loc.y ].food != NO_FOOD )
              statement( tree->left );
              else
                  statement( tree->right );
            break;
        case IF_AT_NEST :
            if ( map[ ants[ant].loc.x ][ants[ant].loc.y ].object == NEST )
              statement( tree->left );
              else
                  statement( tree->right );
            break;
        case IF_CARRYING_FOOD :
/*
           if (( ants[ant].foodCarried != NO_FOOD ) && ( foodId[ ants[ ant
].foodCarried ].numAntsLifting ==
                                                    foodId[ ants[ ant
].foodCarried ].weight ))
*/
            if ( ants[ant].foodCarried != NO_FOOD )
              statement( tree->left );
              else
                  statement( tree->right );
            break;
        case IF_ANOTHER_ANT_HERE :
```

```
          for ( cond = 0, i = 0; (( i < NUM_ANTS ) && ( cond < 2 )); i++ )
            if (( ants[ ant ].loc.x == ants[ i ].loc.x ) && ( ants[ ant
].loc.y == ants[ i ].loc.y ) &&
                                        ( ants[ i ].alive ))
                cond++;
          if ( cond >= 2 )
            statement( tree->left );
            else
                statement( tree->right );
          break;
      case IF_MOVE_INTO_WATER :
          if ( move_into_water() )
            statement( tree->left );
            else
                statement( tree->right );
          break;
      case IF_CANT_LIFT_FOOD :
          if ( ants[ ant ].foodCarried != NO_FOOD )
            if (( foodId[ ants[ ant ].foodCarried ].numAntsLifting <
                                foodId[ ants[ ant ].foodCarried ].weight
))
                statement( tree->left );
            else
                statement( tree->right );
          else
            statement( tree->right );
          break;
      case IF_TIRED_OF_WAITING :
          if (( WAIT_ODDS == 0 ) ||
                  (( ants[ant].foodCarried != NO_FOOD ) &&
                  ( map[ ants[ant].loc.x ][ ants[ant].loc.y ].food ==
ants[ant].foodCarried ) &&
                  ( lrand48() % WAIT_ODDS == 0 ))) {
            statement( tree->left );
          } else
                statement( tree->right );
          break;
        default :
          printf("Illegal type in statement(): %d\n", tree->type );
          treeDump( atree );
          ERROR( E_ILLEGAL_INSTRUCTION );
          break;
    }
}

int numNodes( tree )
TreePtr tree;
{
    if ( tree != NULL_TREE ) {
      nodes++;
      numNodes( tree->left );
      numNodes( tree->right );
    }
}

void spreadPheromone()
{
    int tmpPheromone[ MAPX ][ MAPY ];
    int i, j, pher;
```

```
    for ( j = 0; j < MAPY; j++ )
      for ( i = 0; i < MAPX; i++ )
          tmpPheromone[ i ][ j ] = 0;

    for ( j = 0; j < MAPY; j++ )
      for ( i = 0; i < MAPX; i++ ) {
          pher = map[ i ][ j ].pheromone;
          if ( pher ) {
            tmpPheromone[ i ][ j ] += pher * ( 100 - PHEROMONE_SPREAD ) /
100 ;
            if ( j > 0 )
               tmpPheromone[ i ][ j-1 ] += pher * PHEROMONE_SPREAD / 400 ;
            if ( j < MAPY-1 )
               tmpPheromone[ i ][ j+1 ] += pher * PHEROMONE_SPREAD / 400 ;
            if ( i > 0 )
               tmpPheromone[ i-1 ][ j ] += pher * PHEROMONE_SPREAD / 400 ;
            if ( i < MAPX-1 )
               tmpPheromone[ i+1 ][ j ] += pher * PHEROMONE_SPREAD / 400 ;
          }
      }

    for ( j = 0; j < MAPY; j++ )
      for ( i = 0; i < MAPX; i++ )
          map[ i ][ j ].pheromone = tmpPheromone[ i ][ j ];
}

int findFitness( foodLeft, foodNotFound, numTurns, numNodes )
int foodLeft;
int foodNotFound;
int numTurns;
int numNodes;
{
      return( foodLeft * FOOD_LEFT_WEIGHT +
              foodNotFound * FOOD_UNFOUND_WEIGHT +
              numTurns * TIME_WEIGHT +
              ((int) ( numNodes / NODE_GROUP )) * NODE_WEIGHT );
}

void setUpRun()
{
    int i,j;

    if ( INTERP_SEED )
    {
      srand48( INTERP_SEED );
    }

    for ( j = 0; j < MAPY; j++ )
      for ( i = 0; i < MAPX; i++ ) {
          map[ i ][ j ].object = MAP[ i ][ j ].object;
          map[ i ][ j ].food = MAP[ i ][ j ].food;
          map[ i ][ j ].pheromone = 0;
      }

    for ( ant = 0; ant < MAX_ANTS; ant++ ) {
      ants[ant].loc.x = STARTX;
      ants[ant].loc.y = STARTY;
      ants[ant].dir = lrand48() % 4;
```

```
      ants[ant].foodCarried = NO_FOOD;
      ants[ant].alive = 1;
    }

    for ( i = 0; i < MAX_FOOD; i++ ) {
      foodId[i].weight = FOODID[i].weight;
      foodId[i].numAntsLifting = 0;
      foodId[i].found = 0;
    }

    turns = 0;
    food_in_nest = 0;
}

void setUpFollowFood()
{
    int i;

    for ( i = 0; i < MAX_FOOD; i++ ) {
      foodId[i].moved = 0;
      foodId[i].foundThisTurn = 0;
    }
}

void moveAntFollowFood()
{
    if ( ! foodId[ ants[ ant ].foodCarried ].foundThisTurn ) {

      ants[ ant ].loc.x = ants[ foodId[ ants[ ant ].foodCarried ].leader
].loc.x;
      ants[ ant ].loc.y = ants[ foodId[ ants[ ant ].foodCarried ].leader
].loc.y;
      ants[ ant ].dir   = ants[ foodId[ ants[ ant ].foodCarried ].leader
].dir;

      if ( ants[ foodId[ ants[ ant ].foodCarried ].leader ].foodCarried
== NO_FOOD ) {
          foodId[ ants[ ant ].foodCarried ].numAntsLifting--;
          ants[ ant ].foodCarried = NO_FOOD;
      }
    }
}

void startAnt( tree )
TreePtr tree;
{
    if ( ants[ ant ].foodCarried != NO_FOOD )
      if ( foodId[ ants[ ant ].foodCarried ].moved )
          moveAntFollowFood();
      else
          statement( tree );
    else
      statement( tree );
}

int interp( person )
ProgramType *person;
{
    int i,j;
```

```
    TreePtr tree;
    long nextSeed;

    tree = person->tree;
    atree = tree;

    if ( tree == NULL_TREE )
      ERROR( E_NULL_TREE );

    if ( INTERP_SEED )
    {
      nextSeed = lrand48();
    }

    setUpRun();

    loops = 0;
   while ( ( food_in_nest < NUM_FOOD ) && ( turns < MAX_TURNS*NUM_ANTS )
&& ( loops++ < MAX_TURNS )) {
      if ( PHEROMONE_SPREAD != 0 )
          spreadPheromone();
      setUpFollowFood();
      for ( ant = 0; ant < NUM_ANTS; ant++ )
          if ( ants[ ant ].alive )
            startAnt( tree );
    }

    nodes = 0;
    numNodes( tree );
    person->foodLeft = NUM_FOOD - food_in_nest;

    if ( person->foodLeft != 0 )
      person->numTurns = MAX_TURNS;
    else
      person->numTurns = turns/NUM_ANTS;

    person->numNodes = nodes;

    person->foodNotFound = 0;
    for ( i = 0; i < NUM_FOOD; i++ )
      if ( ! foodId[ i ].found )
          person->foodNotFound++;

  person->fitness = findFitness( person->foodLeft, person->foodNotFound,
person->numTurns, person->numNodes );

    if ( INTERP_SEED )
    {
      srand48( nextSeed );
    }
}
```

```
/*
**
** map.c
**
** this will read in the map file and put it into a 2d array
**
*/

#include <stdio.h>
#include "types.h"
#include "dir.h"
#include "params.h"
#include "error.h"

extern FILE *fpi;

MapType MAP[ MAPX ][ MAPY ];
FoodIDType FOODID[ MAX_FOOD ];
int NUM_FOOD;
int NESTX;
int NESTY;

int readMap()
{
    int i, j;
    char ch;
    char fileName[100];
    char str[ MAPX ];

    sprintf(fileName, "%s%d", MAP_FILE, MAP_VERSION );

    if (( fpi = fopen(fileName, "r")) == NULL ) {
      printf("Error opening map file %s\n", fileName);
      return( 0 );
    }

    NUM_FOOD = 0;
    NESTX = -1;

    for ( j = 0; j < MAPY; j++ ) {
      fgets( str, MAPX+1, fpi );
      for( i = 0; i < MAPX; i++ ) {
          ch = str[i];
          MAP[i][j].object = NOTHING;
          MAP[i][j].food = NO_FOOD;
          switch ( ch ) {
            case ' ' : MAP[i][j].object = NOTHING;
                       break;
            case 'B' : MAP[i][j].object = WALL;
                       break;
            case 'W' : MAP[i][j].object = WATER;
                       break;
            case 'N' : if ( NESTX != -1 )
                          ERROR( E_TWO_NEST );
                       MAP[i][j].object = NEST;
                       NESTX = i;
                       NESTY = j;
                       break;
            case '1' :
```

```
            case '2' :
            case '3' :
            case '4' :
            case '5' :
            case '6' :
            case '7' :
            case '8' :
            case '9' : MAP[i][j].food = NUM_FOOD;
                       FOODID[ NUM_FOOD ].weight = ch - '0';
                       NUM_FOOD++;
                       break;
            default  : printf("Illegal character in map = %d\n", ch);
                       printf("Line %d, char %d\n", j, i);
                       ERROR( E_MAP_FILE );
          }
        }
        do {
            ch = getc( fpi );
        } while ( ch != 10 );
      }
      return(1);
}
```

```
/*
** mutate.c
**
** the mutate() function will modify the binary tree MUTATE percent of the
time
**
*/

#include <math.h>
#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

mutate( person )
ProgramType *person;
{
    ProgramType oldPerson;
    TreePtr tree = person->tree;
    int rnd;
    int mutateOdds;
    int oldType;

    if ( ( lrand48() % MUTATE_ODDS ) == 0 ) {
      mutateOdds = (int) log( (double) person->numNodes ) / log(2) + 1;
      if ( mutateOdds < 2 )
          mutateOdds = 2;
      while ( ( lrand48() % mutateOdds ) && ( tree->left != NULL_TREE ) ) {
          rnd = lrand48() % 2;
          switch ( rnd ) {
            case 0 :
                if ( tree->left != NULL_TREE )
                  tree = tree->left;
                break;
            case 1 :
                if ( tree->right != NULL_TREE )
                  tree = tree->right;
                break;
          }
      }

      oldType = tree->type;

      oldPerson.fitness = person->fitness;
      oldPerson.foodNotFound = person->foodNotFound;
      oldPerson.foodLeft = person->foodLeft;
      oldPerson.numTurns = person->numTurns;

      if ( isLeaf( tree->type ))
          tree->type = getLeafStmt();
      else if ( isNonLeaf( tree->type ))
          tree->type = getNonLeafStmt();
      else
          printf("Illegal type in mutate(): %d", tree->type);

      interp( person );

      if ((( lrand48() % 100 ) < PERCENT_GREEDY_MUTATE ) && ( person->fit-
ness > oldPerson.fitness )) {
```

```
            tree->type = oldType;
            person->fitness = oldPerson.fitness;
            person->foodNotFound = oldPerson.foodNotFound;
            person->foodLeft = oldPerson.foodLeft;
            person->numTurns = oldPerson.numTurns;
        }
    }
}
```

```
/*
** pop.c
**
** find the parents and location of the chlidren using tournament selec-
tion
** update the best fitness
*/

#include "params.h"
#include "types.h"
#include "commands.h"
#include "error.h"

extern int *hood;
extern int *pair;
extern PopType population;

makeChildren( bestIndividual, bestFitness )
int   *bestIndividual ;
int   bestFitness ;
{
    int i, j, best, temp;

    for ( i = 0 ; i < POPULATION_SIZE ; i++ )
      pair[ i ] = 0;

    for ( i = 0 ; i < TOURNAMENT_SIZE ; i++ ) {/* pick tournament */
      do {
          hood[ i ] = lrand48() % POPULATION_SIZE ;
      } while ( pair[ hood[ i ] ] != 0 );
      pair[ hood[ i ] ] = 1;
    }

    for ( i = 0 ; i < TOURNAMENT_SIZE-1 ; i++ ) {/* sort tournament */
      best = i ;
      for ( j = i + 1 ; j < TOURNAMENT_SIZE ; j++ )
          if ( population[ hood[ j ] ].fitness <
                              population[ hood[ best ] ].fitness )
            best = j ;
      if ( best != i ) {
          temp = hood[ i ] ;
          hood[ i ] = hood[ best ] ;
          hood[ best ] = temp ;
      }
    }

    crossover( population[ hood[ 0 ] ],
             population[ hood[ 1 ] ],
            &population[ hood[ TOURNAMENT_SIZE-1 ] ],
            &population[ hood[ TOURNAMENT_SIZE-2 ] ] ) ;

    interp( &( population[ hood[ TOURNAMENT_SIZE-1 ] ] ));
    interp( &( population[ hood[ TOURNAMENT_SIZE-2 ] ] ));

    mutate( &population[ hood [ TOURNAMENT_SIZE-1 ] ] );
    mutate( &population[ hood [ TOURNAMENT_SIZE-2 ] ] );

    dprintf("*****\n");
```

```
    dprintf("parent1.fitness = %d, parent2.fitness = %d\n", population[
hood[ 0]].fitness,
                                                    population[ hood[ 1 ] ].fit-
ness);
    dprintf("child1.fitness = %d, child2.fitness = %d\n", population[
hood[ TOURNAMENT_SIZE-1]].fitness,
                                                     population[ hood[
TOURNAMENT_SIZE-2 ] ].fitness);
    if ( DEBUG ) {
      treeDump( population[ hood[ 0 ]].tree );
      treeDump( population[ hood[ 1 ]].tree );
      treeDump( population[ hood[ TOURNAMENT_SIZE-1 ]].tree );
      treeDump( population[ hood[ TOURNAMENT_SIZE-2 ]].tree );
    }

   if ( population[ hood[ TOURNAMENT_SIZE - 1 ] ].fitness < bestFitness )
      *bestIndividual = hood[ TOURNAMENT_SIZE-1 ] ;

   if ( population[ hood[ TOURNAMENT_SIZE - 2 ] ].fitness < bestFitness )
      *bestIndividual = hood[ TOURNAMENT_SIZE-2 ] ;
}
```

```
/*
** save.c
**
** routines to save & load the population
**
*/

#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
#include "params.h"
#include "types.h"
#include "commands.h"
#include "dir.h"
#include "error.h"

extern FILE *fpo;
extern FILE *fpi;
extern int firstUsed;
extern int itteration;
extern PopType  population;
extern TreePtr firstProg;
extern int restoreDone;

extern char END_OF_PROGRAM_CHARACTER;

popSave()
{
    char filename[70];
    int i;

    END_OF_PROGRAM_CHARACTER = '&';

   sprintf( filename, "%s%02d.save.%06d", TEST_FILE, VERSION, itteration
);

    dprintf("Saving population to file %s\n.", filename );

    if (( fpo = fopen( filename, "w")) == 0 ) {
      printf("error opening file %s for write\n", filename);
      ERROR( E_WRITE_SAVE_FILE );
    }

    for (i = 0; i < POPULATION_SIZE; i++ ) {
      stringTreeDumpRecur( population[i].tree, 0, 1 );
      if ( i < POPULATION_SIZE-1 )
          fprintf( fpo, "%c\n", END_OF_PROGRAM_CHARACTER );
    }

    fclose( fpo );
}

popRestore( popSize )
int *popSize;
{
    char filename[70];
    int bestFitness;

    END_OF_PROGRAM_CHARACTER = '&';
```

```
    sprintf( filename, "%s%02d.save.%06d", TEST_FILE, RESTORE_VERSION,
RESTORE_ITERATION );

    dprintf("Restoring population from file %s\n", filename );

    if (( fpi = fopen( filename, "r")) == 0 ) {
      printf("error opening file %s for read.\n", filename);
      ERROR( E_GET_SAVE_FILE );
    }

    restoreDone = 0;

    *popSize = 0;
   while (( !restoreDone ) && ( *popSize < POPULATION_SIZE ) && ( *popSize
< RESTORE_AMOUNT ))  {
      dprintf("Restoring program #%d\n", *popSize);

      if (yyparse()) {
          printf("Error parsing save file\n");
          ERROR( E_SAVE_FILE );
      }

      population[*popSize].tree = firstProg;

      dprintf("Intepreting program #%d\n", *popSize);

      interp( &population[ *popSize ] );

      (*popSize)++;
    }

    fclose( fpi );
}
```

```
/*
**
** tree.c
**
** tree functions
*/

#include "params.h"
#include "types.h"
#include "error.h"

void copyTreeRecur( tree1, tree2 )
TreePtr tree1;
TreePtr tree2;
{

    tree2->type = tree1->type;

    if ( tree1->left != NULL_TREE ) {
        tree2->left = (TreePtr) alloMem( sizeof( TreeType ) );
        copyTreeRecur( tree1->left, tree2->left );
    } else
        tree2->left = NULL_TREE;

    if ( tree1->right != NULL_TREE ) {
        tree2->right = (TreePtr) alloMem( sizeof( TreeType ) );
        copyTreeRecur( tree1->right, tree2->right );
    } else
        tree2->right = NULL_TREE;
}

void copyTree( tree1, tree2 )
TreePtr tree1;
TreePtr *tree2;
{
    *tree2 = (TreePtr) alloMem( sizeof( TreeType ) );
    (*tree2)->type = tree1->type;

    if ( tree1->left != NULL_TREE ) {
        (*tree2)->left = (TreePtr) alloMem( sizeof( TreeType ) );
        copyTreeRecur( tree1->left, (*tree2)->left );
    } else
        (*tree2)->left = NULL_TREE;

    if ( tree1->right != NULL_TREE ) {
        (*tree2)->right = (TreePtr) alloMem( sizeof( TreeType ) );
        copyTreeRecur( tree1->right, (*tree2)->right );
    } else
        (*tree2)->right = NULL_TREE;
}
```

```
%{
/*
** input.l
**
** lex for file input
**
*/

#undef input()

#include "commands.h"
#include "error.h"
#include "y.tab.h"

extern char END_OF_PROGRAM_CHARACTER ;
extern char *CommandText[];
extern FILE *fpi;
extern int restoreDone;
int lineNo = 1;
char ch;
char lastCh = ' ';
int diff = 0;
int spaceCnt = 0;
int lastNumTabs = 0;

%}

%p 3000

%%

"TURN LEFT"              { return( L_TURN_LEFT ); }
"TURN RIGHT"             { return( L_TURN_RIGHT ); }
"MOVE FORWARD"           { return( L_MOVE_FORWARD ); }
"MOVE RANDOM"            { return( L_MOVE_RANDOM ); }
"MOVE QUASI RANDOM"      { return( L_MOVE_QUASI_RANDOM ); }
"MOVE TO NEST"           { return( L_MOVE_TO_NEST ); }
"MOVE TO NEST THROUGH WATER"{ return( L_MOVE_TO_NEST_THROUGH_WATER ); }
"MOVE INTO WATER"        { return( L_MOVE_INTO_WATER ); }
"PICK UP FOOD"           { return( L_PICK_UP_FOOD ); }
"DROP FOOD"              { return( L_DROP_FOOD ); }
"RELEASE PHEROMONE"      { return( L_RELEASE_PHEROMONE ); }
"ESCAPE PHEROMONE"       { return( L_ESCAPE_PHEROMONE ); }
"DO NOTHING"             { return( L_NOOP ); }

"IF ( FOOD AHEAD ) THEN"       { return( L_IF_FOOD_AHEAD ); }
"IF ( FOOD LEFT ) THEN"        { return( L_IF_FOOD_LEFT ); }
"IF ( FOOD RIGHT ) THEN"       { return( L_IF_FOOD_RIGHT ); }
"IF ( FOOD ADJACENT ) THEN"    { return( L_IF_FOOD_ADJACENT ); }
"IF ( MOVE TO ADJACENT FOOD ) THEN"{ return( L_IF_MOVE_TO_ADJACENT_FOOD
); }
"IF ( MOVE TO ADJACENT PHEROMONE ) THEN"  { return(
L_IF_MOVE_TO_ADJACENT_PHEROMONE ); }
"IF ( MOVE TO AWAY PHEROMONE ) THEN"      { return(
L_IF_MOVE_TO_AWAY_PHEROMONE ); }
"IF ( MOVE TO NEST ) THEN"                { return( L_IF_MOVE_TO_NEST );
}
"IF ( MOVE TO NEST THROUGH WATER ) THEN"  { return(
L_IF_MOVE_TO_NEST_THROUGH_WATER ); }
```

```
"IF ( MOVE TO DEAD ANT ) THEN"      { return( L_IF_MOVE_TO_DEAD_ANT ); }
"IF ( WATER AHEAD ) THEN"           { return( L_IF_WATER_AHEAD ); }
"IF ( OBSTACLE AHEAD ) THEN"        { return( L_IF_OBSTACLE_AHEAD ); }
"IF ( FOOD HERE ) THEN"             { return( L_IF_FOOD_HERE ); }
"IF ( AT NEST ) THEN"               { return( L_IF_AT_NEST ); }
"IF ( CARRYING FOOD ) THEN"         { return( L_IF_CARRYING_FOOD ); }
"IF ( ANOTHER ANT HERE ) THEN"      { return( L_IF_ANOTHER_ANT_HERE ); }
"IF ( MOVE INTO WATER ) THEN"       { return( L_IF_MOVE_INTO_WATER ); }
"IF ( CANT LIFT FOOD ) THEN"        { return( L_IF_CANT_LIFT_FOOD ); }
"IF ( TIRED OF WAITING ) THEN"      { return( L_IF_TIRED_OF_WAITING ); }

"ELSE"                              { return( L_ELSE ); }
"{"                                 { return( L_START ); }
"}"                                 { return( L_END ); }

" "                                 ;
"\t"                                ;

\n                                  {
                                    lineNo++;
                                    spaceCnt = 0;
                                    }

.                                   {
                                    printf( "Line %d: Illegal expression in
lex: %s\n", lineNo, yytext);
                                    ERROR( E_LEX );
                                    }

%%

char myinput()
{
      int tmp;

      if (( diff == -1 ) || ( diff == 1 )) {
            diff = 0;
            if ( ch == END_OF_PROGRAM_CHARACTER )
                  return( 0 );
            else
                  if ( ch == EOF ) {
                        restoreDone = 1;
                        return( 0 );
                  } else
                        return( ch );
      } else if ( diff < -1 ) {
            diff++;
            return( CommandText[ END ][0] );
      } else if ( diff > 1 ) {
            diff--;
            return( CommandText[ START ][0] );
      }

      lastCh = ch;
      ch = fgetc( fpi );
      if ( ch == ' ' )
            spaceCnt++;
      else if ( ch == '\t' )
            spaceCnt += 8;
```

```
      else {
            if ( spaceCnt > 1 ) {
                  if (( spaceCnt % 4 ) != 0 ) {
                        printf( "Illegal Number of spaces (%d) in input
File line %d\n", spaceCnt, lineNo);
                        ERROR( E_BAD_INPUT_SPACING );
                  } else {
                        tmp = spaceCnt / 4;
                        diff = tmp - lastNumTabs;
                        lastNumTabs = tmp;
                        if ( diff < 0 ) {
                              return( CommandText[ END ][0] );
                        } else if ( diff > 0 ) {
                              return( CommandText[ START ][0] );
                        } else
                              spaceCnt = 0;
                  }
            } else {
                  spaceCnt = 0;
                  if (( lastCh == '\n' ) && ( lastNumTabs > 0 )) {
                        diff = -lastNumTabs;
                        lastNumTabs = 0;
                        return( CommandText[ END ][0] );
                  }
            }
      }

      if ( ch == END_OF_PROGRAM_CHARACTER )
            return( 0 );
      else
            if ( ch == EOF ) {
                  restoreDone = 1;
                  return( 0 );
            } else
                  return( ch );
}

char input()
{
      char ret;

      ret = myinput();

      if ( ret == 0 )
          dprintf("End of input file\n");
      else if ( ret == '\n' )
          dprintf("line = %d, ch = \\n, cnt = %d, lnt = %d\n", lineNo,
spaceCnt, lastNumTabs );
      else
          dprintf("line = %d, ch = %c , cnt = %d, lnt = %d\n", lineNo,
ret, spaceCnt, lastNumTabs );

      return(ret);
}

void yyerror( s )
char *s;
{
      printf("Line %d: %s at %s\n", lineNo, s, yytext );
```

```
}

int yywrap()
{
      return(1);
}
```

```
%{
/*
** input.y
**
** yacc file for function input
**
**/
#include "y.tab.h"
#include <stdio.h>
#include "commands.h"
#include "types.h"
#include "error.h"

extern TreePtr firstProg;

#define YYSTYPE TreePtr

%}

%start pgm

%token L_IF_FOOD_AHEAD L_IF_FOOD_LEFT L_IF_FOOD_RIGHT L_IF_FOOD_ADJACENT
%token L_IF_MOVE_TO_ADJACENT_FOOD L_IF_MOVE_TO_ADJACENT_PHEROMONE
%token L_IF_MOVE_TO_AWAY_PHEROMONE L_IF_MOVE_TO_NEST
L_IF_MOVE_TO_NEST_THROUGH_WATER
%token L_IF_MOVE_TO_DEAD_ANT L_IF_WATER_AHEAD L_IF_OBSTACLE_AHEAD
L_IF_FOOD_HERE
%token L_IF_AT_NEST L_IF_CARRYING_FOOD L_IF_ANOTHER_ANT_HERE
%token L_IF_MOVE_INTO_WATER L_IF_CANT_LIFT_FOOD L_IF_TIRED_OF_WAITING
%token L_TURN_LEFT L_TURN_RIGHT L_MOVE_FORWARD L_MOVE_RANDOM
%token L_MOVE_QUASI_RANDOM L_MOVE_TO_NEST L_MOVE_TO_NEST_THROUGH_WATER
%token L_MOVE_INTO_WATER L_PICK_UP_FOOD L_DROP_FOOD L_RELEASE_PHEROMONE
%token L_ESCAPE_PHEROMONE L_NOOP L_START L_END L_ELSE


%%    /* rules */

pgm   :     nonLeaf
            {
                  firstProg = $1;
            }
      ;

nonLeaf:    term
            {
                  $$ = (TreePtr) alloMem( sizeof( TreeType ) );
                  $$->type = (int) $1;
                  $$->left = NULL_TREE;
                  $$->right = NULL_TREE;
            }
      |     nonLeaf nonLeaf
            {
                  $$ = (TreePtr) alloMem( sizeof( TreeType ) );
                  $$->type = PROC2;
                  $$->left = $1;
                  $$->right = $2;
            }
      |     nonTerm L_START nonLeaf L_END L_ELSE L_START nonLeaf L_END
            {
```

```
                $$ = (TreePtr) alloMem( sizeof( TreeType ) );
                $$->type = (int) $1;
                $$->left = $3;
                $$->right = $7;
            }
        ;

nonTerm :L_IF_FOOD_AHEAD
                { $$ = IF_FOOD_AHEAD ; }
        |       L_IF_FOOD_LEFT
                { $$ = IF_FOOD_LEFT ; }
        |       L_IF_FOOD_RIGHT
                { $$ = IF_FOOD_RIGHT ; }
        |       L_IF_FOOD_ADJACENT
                { $$ = IF_FOOD_ADJACENT ; }
        |       L_IF_MOVE_TO_ADJACENT_FOOD
                { $$ = IF_MOVE_TO_ADJACENT_FOOD ; }
        |       L_IF_MOVE_TO_ADJACENT_PHEROMONE
                { $$ = IF_MOVE_TO_ADJACENT_PHEROMONE ; }
        |       L_IF_MOVE_TO_AWAY_PHEROMONE
                { $$ = IF_MOVE_TO_AWAY_PHEROMONE ; }
        |       L_IF_MOVE_TO_NEST
                { $$ = IF_MOVE_TO_NEST ; }
        |       L_IF_MOVE_TO_NEST_THROUGH_WATER
                { $$ = IF_MOVE_TO_NEST_THROUGH_WATER ; }
        |       L_IF_MOVE_TO_DEAD_ANT
                { $$ = IF_MOVE_TO_DEAD_ANT ; }
        |       L_IF_WATER_AHEAD
                { $$ = IF_WATER_AHEAD ; }
        |       L_IF_OBSTACLE_AHEAD
                { $$ = IF_OBSTACLE_AHEAD ; }
        |       L_IF_FOOD_HERE
                { $$ = IF_FOOD_HERE ; }
        |       L_IF_AT_NEST
                { $$ = IF_AT_NEST ; }
        |       L_IF_CARRYING_FOOD
                { $$ = IF_CARRYING_FOOD ; }
        |       L_IF_ANOTHER_ANT_HERE
                { $$ = IF_ANOTHER_ANT_HERE ; }
        |       L_IF_MOVE_INTO_WATER
                { $$ = IF_MOVE_INTO_WATER ; }
        |       L_IF_CANT_LIFT_FOOD
                { $$ = IF_CANT_LIFT_FOOD ; }
        |       L_IF_TIRED_OF_WAITING
                { $$ = IF_TIRED_OF_WAITING ; }
        ;

term    :       L_TURN_LEFT
                { $$ = TURN_LEFT ; }
        |       L_TURN_RIGHT
                { $$ = TURN_RIGHT ; }
        |       L_MOVE_FORWARD
                { $$ = MOVE_FORWARD ; }
        |       L_MOVE_RANDOM
                { $$ = MOVE_RANDOM ; }
        |       L_MOVE_QUASI_RANDOM
                { $$ = MOVE_QUASI_RANDOM ; }
        |       L_MOVE_TO_NEST
                { $$ = MOVE_TO_NEST ; }
```

```
        |       L_MOVE_TO_NEST_THROUGH_WATER
                        { $$ = MOVE_TO_NEST_THROUGH_WATER ; }
        |       L_MOVE_INTO_WATER
                        { $$ = MOVE_INTO_WATER ; }
        |       L_PICK_UP_FOOD
                        { $$ = PICK_UP_FOOD ; }
        |       L_DROP_FOOD
                        { $$ = DROP_FOOD ; }
        |       L_RELEASE_PHEROMONE
                        { $$ = RELEASE_PHEROMONE ; }
        |       L_ESCAPE_PHEROMONE
                        { $$ = ESCAPE_PHEROMONE ; }
        |       L_NOOP
                        { $$ = NOOP ; }
        ;
%%
```

```
#!/bin/sh

# make-params

# Input is the file "params.d"
# Output are files "params.c" and "params.h"
# Intermediate files are "params.1" thru "params.4"
#           params.1: declaration and initial values for the parameters
#           params.2: declares, entry, file manipulation, params file
reading
#           params.3: identify parameters' names and initialize them
#           params.4: code to echo the parameters after all are set.

cat /dev/null > params.3

cat   > params.2 << EOF

#include<string.h>
#include<ctype.h>
#include<stdio.h>
#include"dir.h"
#include"error.h"

#define_int_"%d"
#define_float_"%f"
#define_double_"%lf"
#define_string_"%s"

#define fig(N,M,T) if(strcmp(key,N)==0)  \\
      { sscanf(strtok(NULL," \t\n"),T,M);} else

params( argc, argv ) int argc; char * argv[];
{ FILE * fp; char line[128], * key, paramDir[100];
EOF

cat   > params.4 << EOF
      /* Conditionally print the parameter values. after they are set. */
      if( ! print_the_params ) return;
EOF

# Process defaultparams information. -------------------------------
if [ -f params.d ]
then
cat params.d | awk '
BEGIN {
            format["int"] = "d"; format["float"] = "f";
            format["string"] = "s"; format["double"] = "f";
            print "/* This is the file params.c */" > "params.1" ;
            print "/* This is the file params.h */" > "params.h" ;
            }

NF==0 { next }

$1~/^#/{ next }

/%/   {
            type = substr( $1, 2 ); len = $2; ty = "_" type "_"
            print "printf( \"%" $1 " " $2 "\\n\" );" >> "params.4"
            next
```

```
            }

      { if( type == "string" )
            {
            print "char" $1 "[" len "]= \"" $2 "\";" >> "params.1" ;
            print "     fig(\"" $1 "\"," $1 "," ty ")" >> "params.3"
            print "extern char" $1 "[" len "];" >> "params.h"
            }
        else
            {
            print type "" $1 "= " $2 ";" >>"params.1" ;
            print "     fig(\"" $1 "\",&" $1 "," ty ")" >> "params.3"
            print "extern " type "" $1 ";" >> "params.h" ;
            }
      }
      {
      print "printf( \"   " $1 "   %" format[type] "\\n\", " $1 ");" >>
"params.4"
      }
`
#-------------------------------------------------------------------
cat   >> params.2 << EOF


if( argc != 2 ) { printf("ant <param file number>\n");
ERROR(E_NO_PARAM_FILE_SPECIFIED); }
sprintf(paramDir, "%s%s", PARAMS_FILE, argv[1] );
fp = fopen(paramDir, "r");
if( fp == NULL ) { printf("Error opening %s\n", paramDir);
ERROR(E_OPEN_PARAM_FILE); }

while( fgets( line, 128, fp ) ) { /* read a line from the file "params" */
      key = strtok( line, " \t" );
      if( '%' == key[0] ) printf( "%s\n", line ); /* comments */
      else if( isalpha( key[0] ) ) {
            if( strcmp( key, "EXIT" )==0 ) break; /* quit processing
params */

EOF
#-------------------------------------------------------------------
cat >> params.3 << EOF

      printf("Unknown key-word: %s\n", line);
} /* end if */ } /* end while */

EOF

cat >> params.4 << EOF
} /* end params */
EOF
#-------------------------------------------------------------------
cat   params.[1234] > params.c; \rm params.[1234]

else
      echo Change defaultparams to params.d
fi
```

```
%int
        debug                   0
        print_the_params        0

        POPULATION_SIZE         1000
        TOURNAMENT_SIZE         4
        FOOD_LEFT_WEIGHT        10000000
        FOOD_UNFOUND_WEIGHT     100000
        TIME_WEIGHT             1
        NODE_WEIGHT             10000
        NODE_GROUP              50
        LEAF_ODDS               9
        NON_LEAF_ODDS           6
        MUTATE_ODDS             400
        PERCENT_GREEDY_MUTATE   100
        MAX_TURNS               100
        NUM_ANTS                10
        GOAL_FITNESS            20
        POP_SEED                0
        INTERP_SEED             0
        PHEROMONE_STRENGTH      100
        PHEROMONE_SPREAD        40
        WAIT_ODDS               50
        AUTO_FOOD_DROP          0
        DEBUG                   0
        SHOW_TIME               1000
        DUMP                    10000
        MAX_ITERATIONS          100000
        RESTORE_ITERATION       0
        RESTORE_VERSION         0
        RESTORE_AMOUNT          0
        INSTRUCTION_VERSION     1
        MAP_VERSION             1
        VERSION                 1
```

```
CC = cc
CFLAGS = -Aa -g
LDLIBS = -lm
LDFLAGS = -L/usr/local/lib -g
INCLUDES = -I/usr/include/Motif1.2 -I/usr/include/X11R5
LIBS = -L /usr/lib/Motif1.2 -lXm -L /usr/lib/X11R5 -lXt -lX11
#LIBS = -lXm -lXt -lX11

EXEC  = ant
DISPLAY= display

OBJ         = params.o interp.o commands.o debug.o allo.o \
                  map.o dump.o tree.o error.o input.o y.tab.o lex.yy.o \
                  crossover.o pop.o init.o mutate.o save.o ant.o
DISPLAYOBJ= params.o interp.o commands.o debug.o allo.o \
                  map.o dump.o tree.o error.o input.o y.tab.o lex.yy.o \
                  execute.o display.o

INC   = params.h types.h commands.h scommands.h map.h dir.h error.h

ALL:  $(EXEC) $(DISPLAY)

params.c: params.d
         make-params

display.o:display.c
      $(CC) $(INCLUDES) -g -c $(CFLAGS) display.c

y.tab.o:yinput.y
      yacc -dl yinput.y
      cc -c y.tab.c

lex.yy.o:linput.l
      lex linput.l
      cc -c lex.yy.c

x.tab.h:y.tab.h
      cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

lex.yy.o:x.tab.h

$(DISPLAY):params.c $(DISPLAYOBJ)
      $(CC) $(CFLAGS) $(INCLUDES) -o $@ $(DISPLAYOBJ) $(LIBS) $(LDFLAGS)
$(LDLIBS)
      mv $(DISPLAY) ../

$(EXEC): params.c $(OBJ) $(INC)
      $(CC) -o $@ $(OBJ) $(LDFLAGS) $(LDLIBS)
      mv $(EXEC) ../

$(DISPLAYOBJ):$(INC)

$(OBJ):$(INC)

clean:
      rm -f $(OBJ)
      rm -f $(DISPLAYOBJ)
      rm -f *.tab.*
      rm -f lex.yy.c
```

```
rm -f $(EXEC) $(DISPLAY)
```

```
0      MOVE RANDOM
1      MOVE QUASI RANDOM
1      MOVE TO NEST
0      MOVE TO NEST THROUGH WATER
0      TURN LEFT
0      TURN RIGHT
0      MOVE FORWARD
1      PICK UP FOOD
0      DROP FOOD
1      RELEASE PHEROMONE
0      MOVE INTO WATER
0      ESCAPE PHEROMONE
0      NOOP

1      PROC2
0      IF ( FOOD AHEAD) THEN
0      IF ( FOOD LEFT ) THEN
0      IF ( FOOD RIGHT ) THEN
0      IF ( FOOD ADJACENT ) THEN
0      IF ( FOOD HERE ) THEN
1      IF ( CARRYING FOOD ) THEN
0      IF ( AT NEST ) THEN
1      IF ( MOVE TO ADJACENT FOOD ) THEN
1      IF ( MOVE TO AWAY PHEROMONE ) THEN
0      IF ( MOVE TO DEAD ANT ) THEN
0      IF ( MOVE TO ADJACENT PHEROMONE ) THEN
0      IF ( MOVE TO NEST ) THEN
0      IF ( MOVE TO NEST THROUGH WATER ) THEN
0      IF ( WATER AHEAD ) THEN
0      IF ( OBSTACLE AHEAD ) THEN
0      IF ( ANOTHER ANT HERE ) THEN
0      IF ( CANT LIFT FOOD ) THEN
0      IF ( TIRED OF WAITING ) THEN
```

N

```
111
111
```

```
1111
1111
1111
1111
```

```
111
111
111
```